Approximation Algorithms using Allegories and Coq

Durjay Chowdhury

Supervisor:
Dr. Michael Winter

Submitted in partial fulfilment
of the requirements for the degree of
Master of Science

Department of Computer Science
Brock University
St. Catharines, Ontario

Abstract

In this thesis, we implement several approximation algorithms for solving optimization problems on graphs. The result computed by the algorithm may or may not be optimal. The approximation factor of an algorithm indicates how close the computed result is to an optimal solution. We are going to verify two properties of each algorithm in this thesis. First, we show that the algorithm computes a solution to the problem, and, second, we show that the approximation factor is satisfied. To implement these algorithms, we use the algebraic theory of relations, i.e., the theory of allegories and various extension thereof. An implementation of various kinds of lattices and the theory of categories is required for the declaration of allegories. The programming language and interactive theorem prover Coq is used for the implementation purposes. This language is based on Higher-Order Logic (HOL) with dependent types which support both reasoning and program execution. In addition to the abstract theory, we provide the model of set-theoretic relations between finite sets. This model is executable and used in our examples. Finally, we provide an example for each of the approximation algorithm.

Acknowledgements

It is my pleasure to express my profound gratitude to my supervisor, Dr. Michael Winter not only for his supervision but also for the advice, motivation, and encouragement from the beginning of this research work. I am fortunate enough to have Dr. Winter as my supervisor, as his constant support, guiding and efforts to clarify concepts help me to overcome all obstacles and finish this thesis work smoothly. Without help from him, it would be impossible for me to finish this thesis work.

I would like to thank my supervisory committee members for their support and helpful suggestions. Their valuable comment helped me to draw a proper path for this thesis. The academic and financial assistance that I got from the Brock University Computer Science Department assisted me to complete my degree. I am grateful for this.

Finally, I want to thank all of my family members and friends for their support and encouragement.

Contents

List of Figures

Chapter 1

Introduction

An optimization problem is a problem in which one tries to find the best among the feasible solutions to a problem. As an example consider the traveling salesman problem. This problem consists of a number of cities and distances among those. A solution to the problem is a tour through all cities. Obviously, one is interested in the best solution, i.e., a tour with the least overall distance to travel. Often it is not feasible to find an optimal solution of an optimization problem. For example, the traveling salesman problem is known to be NP-complete so that computing an optimal solution for a large input, i.e., a large net of cities, may need more time than available.

Approximation algorithms are used to find the feasible solution for an optimization problem. This solution may or may not be an optimal one. The ratio between the result computed by the approximation algorithm and the optimal solution is called the approximation factor. The approximation factor tells us how close the result that we get using the algorithm is to an optimal solution.

The goal of this thesis is to construct a common framework for the implementation and verification of approximation algorithms on graphs. Using the framework we will implement several approximation algorithms and show that our implementation is logically correct, i.e., that the algorithm terminates and produces a solution to the problem. Furthermore, we will also verify the approximation factor of the algorithm formally.

Verification of the software is an important area of computer science. The method works by providing a formal proof that a program satisfies all properties that are supposed to be satisfy. In the case of approximation algorithms the approximation factor is one of the essential properties of the algorithm that needs to be verified. In this thesis we use the

abstract theory of binary relations to models graphs and to verify the required properties of the algorithms. Concretely, we will use a categorical approach using allegories and various extension thereof. In order to handle approximation factors we will use an abstract cardinality function on binary relations, i.e., a function that assigns to every element of an allegory, i.e, every relation, an element of a suitable monoid.

Since one of our goals is to perform formal reasoning about programs, we need a language that provides means for verification. Unfortunately, most of the general purpose programming languages do not support formal reasoning about programs. On the other hand, some functional programming languages, in particular languages that are based on type theory encoding Higher-Order Logic (HOL), are rich enough for programming as well as verification. The programming language Coq [20] is one of the examples of such languages. In our application, we will use this language. Coq is a French-developed proof assistant implementing a functional programming language and tactic-based theorem proving. In Coq, programming and verification is possible using the same language, which is the primary advantage of that language. This language is very popular among mathematicians and computer scientists as it supports natural deduction style reasoning.

First, we define the abstract theory of allegories and its various extensions including cardinality functions. Then we provide a concrete model of this theory by implementing set-theoretic relations between finite sets. This model also serves as the basis for executing our algorithms on concrete examples. We also show some basic properties of relations within that theory. Each approximation algorithm is then defined and verified using the abstract theory. The main advantage of this approach is that the assumptions for an algorithm to work correctly become very apparent. As a further consequence, each algorithm is correct for every model of the theory including the aforementioned model of binary relations between finite sets.

Besides the main goal of the thesis it is interesting to discuss some mathematical relationships within the theory of allegories, its implementation in type-theory, and their relationship to programming languages. We will often refer to these issues during the planning and development of the framework and the implementation of the algorithms.

This thesis is not the first attempt to use relation-algebraic methods for approximation algorithms [1]. There are at least two major differences between the approach taken in [1] and this thesis. First of all, this thesis takes a more abstract approach by only requiring prop-

erties that are needed for the algorithm at hand. For example, [1] assumes the so-called Tarski-rule and the point axiom as basic axioms. This thesis does not use the point axiom at all and any usage of the Tarski-rule is made explicit whenever it is needed. In addition, in [1] the cardinality function is assumed to return a natural number. As a consequence, the order on cardinalities is linear, subtraction is available and all relations are finite. In particular, any recursion or loop based on removing a pair from a relation in each iteration will terminate. In this thesis the cardinality function returns a value from an arbitrary ordered monoid. This includes non-linear ordered monoids, monoids without subtraction, and does not imply that relations are finite. As a consequence we make explicit whenever finiteness is required by explicitly assuming that the inclusion order on relations is well-founded. Secondly, [1] uses a imperative language, i.e., loops, to implement the algorithms. An implementation of the Floyd-Hoare calculus is used to show partial correctness. Termination is shown as a separate theorem. This thesis uses the internal functional language of Coq to implement the algorithms. Obviously, this implies that the programs are recursive. In addition, since all Coq programs need to terminate, we have to prove termination as an integral part of the implementation and not as a separate theorem. Last but not least, we want to mention that some algorithms in [1] compute auxiliary results such as a matching, that are convenient to use in the correctness proofs but definitely not necessary. The algorithms in this thesis avoid computing these values.

The implementation of our framework for categories of relations is different from the one that developed by Damien Pous [23]. His library does not use the class system of Coq in order to implement hierarchies of algebras, categories and theories. He declared all operations and constants, i.e., meet, join, converse, complement, residual etc., at once as a record structure in Coq. This structure has a bit string like parameter that can be used to select a subset of the operations to work with, i.e., the hierarchy of structure is encoded using the bit string. During the declaration of a particular property, the bit string has to be chosen to include those operators which are used in the property together with their axioms selected from a similar record with the same bit string. In our implementation, we strongly rely on the class hierarchy of Coq, which means that we only declare those operators and axioms which are needed by that particular property or structure. The main advantage of using classes is that it gives a direct representation of the hierarchy of the mathematical structures in question.

The thesis is organized as follows. Before addressing details of the implementation of several algorithms, we will discuss mathematics preliminaries in Chapter 2. Then we will introduce categories and allegories in Chapter 3. In Chapter 4, we talk about approximation algorithms, approximation factors, and solutions of approximation problems. The following chapter will be about the programming language Coq and its functionality. Details about the development of a common framework, the implementation of several approximation algorithms and their proof of correctness will follow in Chapter 6 and 7. An example for each approximation problem that uses the concrete model of set-theoretic relations between finite sets is provided in Chapter 7. Finally, in Chapter 8, we will give concluding remarks and some basic outlines for the future work.

Chapter 2

Preliminaries

In this section we define the basic concepts such as lattices and set-theoretic relations that are needed throughout the thesis.

2.1 Partially-Ordered Sets

A (partially) ordered set is a very basic concept within mathematics and computer science. It formalizes the idea that some elements are smaller than others.

Definition 2.1.1. *A binary relation $\leqslant$ is called an ordering (or an order relation) on a set A iff*[1] *for all $a, b, c \in A$ we have,*

Reflexive : $a \leqslant a$

Antisymmetric : *if* $a \leqslant b$ *and* $b \leqslant a$ *then* $a = b$

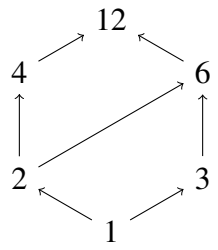
Transitive : *if* $a \leqslant b$ *and* $b \leqslant c$ *then* $a \leqslant c$

The pair $(A, \leqslant)$ is called a poset.

We will visualize a finite ordered set usually by its Hasse-Diagram (see example below). In such diagram a line that goes upward from an element y to an element x indicates that y is strictly smaller than x, i.e., $y \leqslant x$ and $y \neq x$, and there is no element between the two.

[1] We use the abbreviation iff for if and only if.

Example 2.1.1. *Suppose we have D = {1, 2, 3, 4, 6, 12}. The order relation on D is given by the property of dividing evenly. For example,* 2 *divides* 4 *evenly so that* $2 \leqslant 4$ *but neither does* 6 *divide* 4 *nor does* 4 *divide* 6 *evenly. The ordering is visualized in below as a Hasse-Diagram.*



In the remainder of this section we will use the example above to illustrate important concepts.

2.1.1 Upper and Lower Bounds

Upper bounds of a subset of elements are based on the concept of being greater or equal than all elements of the subset. We obtain the following definition.

Definition 2.1.2. *Let* $(L, \leqslant)$ *be a poset and* $A \subseteq L$*. Then an upper for A is an element* $u \in L$ *so that* $x \leqslant u$ *for all* $x \in A$*.*

Note that a subset does not necessarily have upper bounds. In addition, it may have more than one upper bound as the following example shows.

Example 2.1.2. *Upper bounds not need to be unique. There may be more than one upper bound. In Figure 2.1, the upper bounds of {2,3} are 6 and 12. On the other hand, the set {4.6} has only one upper bound, the element {12}.*

Lower bounds are defined dually, i.e., they are upper bounds of the reversed order.

Definition 2.1.3. *Let* $(L, \leqslant)$ *be a poset and* $A \subseteq L$*. Then an lower for A is an element* $u \in L$ *so that* $u \leqslant x$ *for all* $x \in A$*.*

Example 2.1.3. *Similar to upper bounds, lower bounds also need not to be unique. In Figure 2.1, the lower bound of 12 will be {1, 2, 3, 4, 6} and the lower bound of {2, 3} will be {1}.*

2.1.2 Greatest and Least Element

Besides upper bounds a subset may provide a greatest element.

Definition 2.1.4. *Let $(L, \leqslant)$ be a poset and $A \subseteq L$. Then a greatest element of A is an upper bound g of A so that $g \in A$.*

Again, a greatest element may not exist.

Example 2.1.4. *Greatest elements may not exist but if they do, they are unique. In Figure 2.1, the set $\{4,6\}$ does not have a greatest element, and the greatest element of $\{1,2,3,6\}$ is 6.*

Similar to lower bounds a least element is dually defined to a greatest element.

Definition 2.1.5. *Let $(L, \leqslant)$ be a poset and $A \subseteq L$. Then a least element of A is an lower bound g of A so that $g \in A$.*

Example 2.1.5. *Similar to greatest elements least elements are also unique. In Figure 2.1, the set $\{4,6\}$ does not have a least element, and the least element of $\{1,2,3,6\}$ is 1.*

2.1.3 Join and Meet

Among the upper bounds the least upper bound might be of interest.

Definition 2.1.6. *Let $(L, \leqslant)$ be a poset and $A \subseteq L$. The least upper bound or join for A is the least element of the set of upper bounds of A.*

Example 2.1.6. *Consider the set $\{2,3\}$ of the example above. The set of upper bounds is $\{6,12\}$ among which 6 is the smallest, i.e., 6 is the least upper bound of $\{2,3\}$.*

The least element of poset will be the join of the empty subset and the least upper bound of the whole poset will be the greatest element of a poset if it exists. The join is denoted by $\bigsqcup S$. If S is a set with two elements, i.e., if $S = \{x, y\}$, then we write $x \sqcup y$ instead of $\bigsqcup S$. In this case we have $z = x \sqcup y$ iff $x \leqslant z$ and $y \leqslant z$ and if $x \leqslant v$ and $y \leqslant v$, then $z \leqslant v$.

Dually, we define greatest lower bounds.

Definition 2.1.7. *Let $(L, \leqslant)$ be a poset and $A \subseteq L$. The greatest lower bound or meet for A is the greatest element of the set of lower bounds of A.*

Example 2.1.7. *Again, let us consider the set $\{2,3\}$. The set of lower bounds is $\{1\}$ so that 1 is the greatest lower bound of $\{2,3\}$.*

$\bigsqcap S$ denotes the meet. If S is a set with two elements, i.e., if $S = \{x, y\}$, then we write $x \sqcap y$ instead of $\bigsqcap S$. In this case we have $z = x \sqcap y$ iff $x \geqslant z$ and $y \geqslant z$ and if $x \geqslant v$ and $y \geqslant v$, then $z \geqslant v$.

2.1.4 Upper and Lower Semilattices

An upper semilattice is a poset which has a join for every pair of elements. Alternatively, an upper semilattice can be defined algebraically as follows. [3]

Definition 2.1.8. *A structure $(L, \sqcup)$ is a upper semilattice if and only if for all x, y and z in L,*

1. $x \sqcup y = y \sqcup x$ *(commutativity)*
2. $(x \sqcup y) \sqcup z = x \sqcup (y \sqcup z)$ *(associativity)*
3. $x \sqcup x = x$ *(idempotency)*

A lower semilattice is poset which has a meet for any pair of elements. It can also be defined algebraically [3].

Definition 2.1.9. *A structure $(L, \sqcap)$ is a lower semilattice if and only if for all x, y and z in L,*

1. $x \sqcap y = y \sqcap x$ *(commutativity)*
2. $(x \sqcap y) \sqcap z = x \sqcap (y \sqcap z)$ *(associativity)*
3. $x \sqcap x = x$ *(idempotency)*

2.2 Lattices

According to order theory, a lattice is a poset where for every two elements there exists a unique least upper bound and a greatest lower bound.
The algebraic definition of the lattice is given below:

Definition 2.2.1. *A structure $(L, \sqcap, \sqcup)$ is a lattice if and only if, L is both upper semilattice and lower semilattice and also for all x, y in L,*

$x \sqcap (x \sqcup y) = x \quad$ *and* $\quad x \sqcup (x \sqcap y) = x$ *(absorption)*

Both the algebraic and order-theoretic definition are equivalent [3]. Any one of the definitions can be used depending on which one is more convenient. Using the algebraic definition the order of the lattice can be defined by $x \leqslant y \iff x \sqcap y = x$ for all $x, y \in L$.

2.2.1 Distributive Lattice

A lattice is a distributive lattice if both operations join and meet distribute over each other.

Definition 2.2.2. *L is a distributive lattice if following properties holds for all x, y, z in L,*

$$x \sqcup (y \sqcap z) = (y \sqcup z) \sqcap (y \sqcup z)$$

We can also distribute meet operation over the join operation which is an exact dual of our defined one. Both of those properties can obtain from one another.

2.2.2 Bounded Lattice

A bounded lattice is a lattice that has a greatest and a least element denoted by 1 and 0, respectively. The algebraic definition is as follows.

Definition 2.2.3. *A lattice L is a bounded lattice if for all x in L,*

$$x \sqcup 0 = x \quad \textit{and} \quad x \sqcap 1 = x$$

A Lattice L is a bounded distributive lattice if it is a distributive and bounded lattice.

Definition 2.2.4. *A Lattice L with* 0 *is 0-distributive if for all x, y, z in L,*

$$(x \wedge y) = 0 \sqcap (x \wedge z) = 0 \Rightarrow x \wedge (y \vee z) = 0$$

Definition 2.2.5. *A Lattice L with* 1 *is 1-distributive if for all x, y, z in L,*

$$(x \vee y) = 1 \sqcap (x \vee z) = 1 \Rightarrow x \vee (y \wedge z) = 1$$

If a lattice is both 0-distributive and 1-distributive is called 0-1 distributive lattice [13].

2.3 Heyting Algebras

In this section, we discuss a different class of lattice called Heyting algebras. We use additional binary implication operation for defining Heyting algebras. This implication operation is denoted by $\rightarrow$. A Heyting algebra is a bounded lattice equipped with this binary operation. This binary implication also represents a weak form of complementation which is known as relative pseudo-complement.

The formal definition of Heyting algebra is given below:

Definition 2.3.1. *A Heyting algebra is a bounded lattice L with a binary operation* $\rightarrow$ *so that for all* $x, y, z \in L$ *and* x, y, z *in L,*

1. $x \rightarrow x = 1$

2. $x \sqcap (x \to y) = x \sqcap y$
3. $y \sqcap (x \to y) = y$
4. $x \sqcap (y \to z) = (x \to y) \sqcap (x \to z)$

According to [19], the implication operation of a Heyting algebra also be characterized by the equivalence, $z \sqsubseteq x \to y \iff x \sqcap z \sqsubseteq y$ for all x, y, z *in* L,

Heyting algebras are less often called pseudo-Boolean algebras, or even Brouwer lattices [19]. As lattices, Heyting algebras are distributive.

Theorem 2.3.1. *Every Heyting algebra is distributive.*

Note that a finite lattice is always complete so that joins distribute over arbitrary meets and vice versa in a finite Heyting algebra.

A Heyting algebra gives rise to a pseudo-complement operation defined as $\overline{x} := x \to 0$. This element can be characterized by $x \sqcap y = 0$ iff $y \sqsubseteq \overline{x}$. In particular, we have $x \sqcap \overline{x} = 0$. On the other hand, $x \sqcup \overline{x} = 1$ might not be true.

2.4 Boolean Algebras

A Boolean algebra is a complemented distributive lattice. This type of structure hold essential properties both for logic and set operation. Its elements can be viewed as a generalization of truth tables. Boolean algebras are also a particular case of de Morgan algebras. The formal definition of Boolean algebra is given below:

Definition 2.4.1. *A Boolean operation algebra is a Heyting algebra with,*

$x \sqcup \overline{x} = 1$ *for all elements* $x \in L$

As every finite Boolean algebra is isomorphic to a lattice of a subset of a finite set, the number of elements for every finite Boolean algebra is a power of two. So any poset with a different number of elements is not a Boolean algebra.

2.5 Set-theoretic Relations

A relation defines a connection or a relationship between elements. In mathematics, a relation is a set of ordered pairs defining a relationship between the elements of each pair. If A is a set and B is another set, then a relation R between them is a subset of $A \times B$, i.e.,

$R \subseteq A \times B$. If $(a,b) \in R$, then will often write aRb where $a \in A$ and $b \in B$. Furthermore, if R is a relation between A and B we will indicate this also by $R : A \to B$.

Example 2.5.1. *In this example we want to define a relation S between seasons and countries. Therefore we define the first set as $SEAS$ = {Summer, Rainy, Fall, Late Autumn, Winter, Spring} and the second set as $CNTRY$ = {Bangladesh, Canada, Colombia}. The relation $S : SEAS \to CNTRY$ indicates in which country a particular season exists:*

S = {(Summer, Bangladesh), (Rainy, Bangladesh), (Fall, Bangladesh), (Winter, Bangladesh), (Late Autumn, Bangladesh), (Spring, Bangladesh), (Summer, Canada), (Fall, Canada), (Winter, Canada), (Spring, Canada), (Summer, Colombia), (Rainy, Colombia)}

2.5.1 Matrix Representation

A relation between two finite sets can be represented by a Boolean matrix (see [15]). Therefore, we assume a linear order on the elements of each set. Usually we use the order given by sequence in which we presented the elements of the corresponding set. If there is a relationship between two elements (a,b), then the matrix will have a 1 (for true) in the row-column entry corresponding to a and b. The entry will be 0 if there is no relationship between the elements. Please note that 0 and 1 are not integers. The operations are based on the Boolean interpretation. This kind of matrix representation is one of the best ways to visualize a relation between the elements of two sets.

For example, we can present the relation S from the previous example as a Boolean matrix as follows:

$$S = \begin{array}{c} \begin{array}{cccccc} Summer & Rainy & Fall & LateAutumn & Winter & Spring \end{array} \\ \begin{pmatrix} 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & 0 & 1 & 0 & 1 & 1 \\ 1 & 1 & 0 & 0 & 0 & 0 \end{pmatrix} \end{array} \begin{array}{c} \\ Bangladesh \\ Canada \\ Colombia \end{array}$$

After removing the label following the convention mentioned above, the actual matrix representation of relation S becomes:

$$S = \begin{pmatrix} 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & 0 & 1 & 0 & 1 & 1 \\ 1 & 1 & 0 & 0 & 0 & 0 \end{pmatrix}$$

This type of representation is convenient when elements of two set are finite and reasonably small in number. We use matrix representation for most of the examples that we discuss later.

2.5.2 Basic operations

Since set-theoretic relations are sets of pairs, the usual set operations such as meet, join and complement are available for relations as well. Because all relations between two sets form a Boolean algebra we will use the notation of Boolean algebras to denote these operations. In terms of matrices, these operations can be performed componentwise by applying the Boolean operations and, or, and not to the corresponding entries in the matrices. For example, if P and Q are the following matrices:

$$P = \begin{pmatrix} 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & 0 & 0 & 1 & 1 & 1 \\ 0 & 0 & 1 & 0 & 1 & 0 \end{pmatrix}, Q = \begin{pmatrix} 1 & 1 & 1 & 0 & 0 & 1 \\ 1 & 1 & 0 & 1 & 0 & 0 \\ 1 & 1 & 1 & 0 & 0 & 1 \end{pmatrix}$$

Then the operation $P \sqcap Q$ will return following matrix:

$$P \sqcap Q = \begin{pmatrix} 1 & 1 & 1 & 0 & 0 & 1 \\ 1 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 \end{pmatrix}$$

The operation $P \sqcup Q$ will return following matrix:

$$P \sqcup Q = \begin{pmatrix} 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 0 & 1 & 1 & 1 \\ 1 & 1 & 1 & 0 & 1 & 1 \end{pmatrix}$$

The operation $\overline{P}$ will return following matrix:

$$\overline{P} = \begin{pmatrix} 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 1 & 0 & 0 & 0 \\ 1 & 1 & 0 & 1 & 0 & 1 \end{pmatrix}$$

We can also express the implication operation ($\rightarrow$) using the operations $\bar{\cdot}$ and $\sqcup$, i.e., we have $P \rightarrow Q = \overline{P} \sqcup Q$. Consequently, the operation $P \rightarrow Q$ will give us the following

matrix.

$$P \to Q = \begin{pmatrix} 1 & 1 & 1 & 0 & 0 & 1 \\ 1 & 1 & 1 & 1 & 0 & 0 \\ 1 & 1 & 1 & 1 & 0 & 1 \end{pmatrix}$$

The empty set is the set that does not contain any elements. The empty relation is the empty set (of pairs). We can represent the empty relation by the constant 0-matrix, i.e., the matrix

$$\begin{pmatrix} 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \end{pmatrix}$$

The universal relation on a set A is the set $A \times A$. In a universal relation, all the pairs of elements are included. The matrix representation of a universal relation will be:

$$\begin{pmatrix} 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 & 1 \end{pmatrix}$$

2.5.3 Converse Relation

Assume two sets A and B and a relation relation R from A to B. The converse of R is denoted by $R^{\smile}$ and a relation from B to A. Its formal definition is as follows:

Definition 2.5.1. *If $R \subseteq A \times B$, then $R^{\smile} := \{(b,a) \mid (a,b) \in R\}$.*

In terms of matrices the converse of R is represented by the transposed matrix of R. For example, the matrix for $P^{\smile}$ is

$$P^{\smile} = \begin{pmatrix} 1 & 1 & 0 \\ 1 & 0 & 0 \\ 1 & 0 & 1 \\ 1 & 1 & 0 \\ 1 & 1 & 1 \\ 1 & 1 & 0 \end{pmatrix}$$

2.5.4 Identity Relation

The identity relation on set A is the set $\{(x, x) \mid x \in A\}$. The matrix representation of the identity relation is given below:

$$\begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

2.5.5 Composition of Relations

Assume three sets X, Y and Z, a relation R from X to Y, and a relation S from Y to Z. The composition of relation R and S, denoted by $R;S$, is defined as follows:

Definition 2.5.2. $R;S := \{(x, z) \mid \exists y : ((x, y) \in R \wedge (y, z) \in S)\}$.

The composition of two relations represented as Boolean matrix can be computed similar to the matrix multiplication known from linear algebra. Instead of multiplication we use the Boolean and, and instead of summation we use the Boolean or.

Let assumes two relations P and Q with

$$P = \begin{pmatrix} 1 & 0 & 1 \\ 0 & 0 & 1 \\ 1 & 0 & 0 \end{pmatrix} \qquad Q = \begin{pmatrix} 0 & 0 & 1 \\ 1 & 0 & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

The operation $P;Q$ will return the following matrix:

$$\begin{pmatrix} 0 & 0 & 1 \\ 0 & 0 & 1 \\ 0 & 0 & 1 \end{pmatrix}$$

Note that each operation of this chapter can effectively be computed if the relations are finite. This is the basis of our implementation of set-theoretic relations between finite sets in Chapter 6.

Chapter 3

Categories and Allegories

In this chapter, we focus on different types of categories and allegories. We also discuss several constructions within categories or allegories that we need for our framework.

3.1 Category Theory

Category theory is an alternative to set theory. It formalizes mathematical structures using the concepts of a collection of objects and arrows. A category has a fundamental composition operation on arrows (or morphisms) that is associative. Beside this, there exists an identity morphism for each object. Morphisms should be considered as maps between mathematical structures. The formal definition of a category is given below:

Definition 3.1.1. *A category C is defined by,*

1. *a collection of objects denoted by* Obj_C,
2. *a collection of morphisms* $C[X,Y]$, *for every pair of objects X and Y,*
3. *an operation* ; *mapping an* f *in* $C[X,Y]$ *and a* g *in* $C[Y,Z]$ *to a morphism* $f;g$ *in* $C[X,Z]$ *which is associative,*
4. *an identity morphism for every object Y denoted by* $\mathbb{I}_Y$, *such that for all* f *in* $C[X,Y]$ *and* g *in* $C[Y,Z]$ *we have* $f;\mathbb{I}_Y = f$ *and* $\mathbb{I}_Y;g = g$.

Some computer scientists and mathematicians are very familiar with categories. They are widely used to describe systems such as databases and models of theoretical physics. Categories of matrices can also be used to obtain an abstract approach to linear algebra.

3.2 Allegories

An allegory is a category where the morphisms are considered to be binary relations. They are an abstraction of the category of set-theoretic relations between sets.

Definition 3.2.1. *A category $\mathcal{R}$ is an allegory satisfying the following,*

1. *Every $\mathcal{R}[A,B]$ is a lower semilattice. $\sqcap$ and $\sqsubseteq$ are use to denote meet and induced ordering respectively. Elements in $\mathcal{R}[A,B]$ are called relations.*
2. *There is a montone operation $^\smile$,i.e, such that for all relations $Q : A \to B$ and $S : B \to C$ the following holds:*
   $(Q;S)^\smile = S^\smile; Q^\smile$ *and* $(Q^\smile)^\smile = Q$.
3. $Q; R \sqcap S \sqsubseteq Q; (R \sqcap Q^\smile; S)$ *for all relations $Q : A \to B, R : B \to C$ and $S : A \to C$.*
4. $Q; (R \sqcap S) \sqsubseteq Q; R \sqcap Q; S$ *for all relations $Q : A \to B$ and $R, S : B \to C$.*

For an allegory, the property $(Q \sqcap R)^\smile = Q^\smile \sqcap R^\smile$ is satisfied. In [19], there are some other properties that can be shown by using the axioms of an allegory.

Lemma 3.2.1. *Let $\mathcal{R}$ be an allegory, A,B,C be objects of $\mathcal{R}$ and $Q, R : A \to B, S : B \to C, T : A \to C$, and $U, V : A \to A$. Then we have*

1. $\mathbb{I}_A{}^\smile = \mathbb{I}_A$,
2. $(Q \sqcap R); S \sqsubseteq Q; S \sqcap R; S$,
3. *For both argument ; is monotone,*
4. $Q; S \sqcap T \sqsubseteq (Q \sqcap T; S^\smile); S$
5. $Q; S \sqcap T \sqsubseteq (Q \sqcap T; S^\smile); (S \sqcap Q^\smile; T)$,
6. $Q \sqsubseteq Q; Q^\smile; Q$,
7. $\mathbb{I}_A \sqcap (U \sqcap V); (U \sqcap V)^\smile = \mathbb{I}_A \sqcap U; V^\smile = \mathbb{I}_A \sqcap V; U^\smile$,
8. $Q = (\mathbb{I}_A \sqcap Q; Q^\smile); Q = Q; (\mathbb{I}_B \sqcap Q^\smile; Q)$.

Some specific relations are defined in [19].

Definition 3.2.2. *Let $\mathcal{R}$ be an allegory where $Q : A \to B$. Then we call*

1. *Q is univalent if and only if $Q^\smile; Q \sqsubseteq \mathbb{I}_B$,*
2. *Q is total if and only if $\mathbb{I}_A \sqsubseteq Q; Q^\smile$,*
3. *Q is map if and only if Q is univalent and total,*
4. *Q is injective if and only if $Q^\smile$ is univalent,*
5. *Q is surjective if and only if $Q^\smile$ is total,*

6. *Q is bijective if and only if $Q^{\smile}$ is map,*
7. *Q is bijection if and only if Q is a bijective map,*
8. *Q is symmetric if and only if $Q = Q^{\smile}$.*

In [19], some interesting properties for univalent relation are shown.

Lemma 3.2.2. *Let $\mathcal{R}$ be an allegory, A,B,C be objects of $\mathcal{R}$ and $Q : A \to B$, $R, S : B \to C, T : C \to A$, and $U : C \to B$. If Q is univalent, then*

1. $Q;(R \sqcap S) = Q;R \sqcap Q;S$,
2. $T;Q \sqcap U = (T \sqcap U;Q^{\smile});Q$.

The dual properties of Lemma 3.2.2 i.e., by reversing the order in the composition, also hold. The following lemma holds for mappings [19].

Lemma 3.2.3. *Let $\mathcal{R}$ be an allegory, A,B,C be objects of $\mathcal{R}$ and $Q : A \to B$, $R : A \to C$, $S : D \to B$ be arbitrary relations and $f : B \to C$ and $g : A \to D$ be mappings. Then we have*

1. $Q;f \sqsubseteq R \quad iff \quad Q \sqsubseteq R;f^{\smile}$,
2. $g^{\smile};Q \sqsubseteq S \quad iff \quad Q \sqsubseteq g;S$,

Definition 3.2.3. *Let $\mathcal{R}$ be an allegory and A an object. A relation $R : A \to A$ is called a partial identity if and only if $R \sqsubseteq \mathbb{I}_A$.*

The following properties hold for partial identities as shown in [19].

Lemma 3.2.4. *Let $\mathcal{R}$ be an allegory, A,B,C objects of $\mathcal{R}$ and $S, T : B \to B$ partial identity $Q, U : A \to B$, and $R, V : B \to C$ arbitrary relation. Then we have*

1. $S^{\smile} = S$,
2. $S;S = S$,
3. $S;T = S \sqcap T$,
4. $Q;(S \sqcap T) = Q;S \sqcap Q;T$ *and* $(S \sqcap T);R = S;R \sqcap T;R$,
5. $(Q \sqcap U);(S \sqcap T) = Q;S \sqcap U;T$ *and* $(S \sqcap T);(R \sqcap V) = S;R \sqcap T;V$.

We use all those lemmas and axioms for proving other lemmas and theorems.

3.3 Distributive Allegories

In a distributive allegory every $\mathcal{R}[A, B]$ is a distributive lattice with a least element. The formal definition is:

Definition 3.3.1. *An allegory $\mathcal{R}$ is a distributive allegory if it satisfies the following:*

1. *Every $\mathcal{R}[A,B]$ is a distributive lattice with least element where we denote union and the least element by $\sqcup$ and $\perp\!\!\!\perp_{AB}$ respectively.*

2. $Q;(R \sqcup S) = Q;R \sqcup Q;S$, *for all* $Q : A \to B$, $R,S : B \to C$.

3. $Q;\perp\!\!\!\perp_{BC} = \perp\!\!\!\perp_{AC}$, *for all* $Q : A \to B$.

According to [19], the following lemma is a consequence of the definition of distributive allegories.

Lemma 3.3.1. *Let $\mathcal{R}$ be a distributive allegory. If $Q,R : A \to B$ and $S : B \to C$, we have,*

1. $\perp\!\!\!\perp^{\smile}_{AB} = \perp\!\!\!\perp_{BA}$,
2. $\perp\!\!\!\perp_{CA};Q = \perp\!\!\!\perp_{CB}$,
3. $(Q \sqcup R)^{\smile} = Q^{\smile} \sqcup R^{\smile}$,
4. $(Q \sqcup R);S = Q;S \sqcap R;S$.

3.4 Division Allegories

According to the hierarchy of allegories, the next step after distributive allegories are division allegories. In the division allegories ; is a lower adjoint.

Definition 3.4.1. *A distributive allegory $\mathcal{R}$ is called division allegory iff for all relations $R : B \to C$ and $S : A \to C$ there is a left residual $S/R : A \to B$ such that for all relation $Q : A \to B$ the following holds:*

$$Q;R \sqsubseteq S \iff Q \sqsubseteq S/R.$$

There also exists an upper right adjoint for ; in a division allegory which is called a right residual. For relations $Q : A \to B$ and $S : A \to C$ the right residual $Q \backslash S$ is defined by $(S^{\smile}/Q^{\smile})^{\smile}$.
A symmetric version of the residuals can be defined as $syQ(Q,R) := (Q\backslash R) \sqcap (Q^{\smile}/R^{\smile})$. The following lemmas were shown in [19]:

Lemma 3.4.1. *Let $\mathcal{R}$ be a division allegory. If $Q,Q_1,Q_2 : A \to B$, $R,R_1,R_2 : B \to C$, and $S,S_1,S_2 : A \to C$, then we have,*

1. $Q \sqsubseteq (Q;R)/R$ *and* $R \sqsubseteq Q\backslash(Q;R)$,

2. $(S/R);R \sqsubseteq S$ *and* $Q;(Q\backslash S) \sqsubseteq S$,

3. $S/(Q\backslash S) \sqsubseteq Q$ *and* $(S/R) \backslash S \sqsubseteq R$,

4. $Q_2 \sqsubseteq Q_1, R_2 \sqsubseteq R_1$ *and* $S_2 \sqsubseteq S_1$ *implies* $S_1/R_1 \sqsubseteq S_2/R_2$ *and* $Q_1 \backslash S_1 \sqsubseteq Q_2 \backslash S_2$,

5. $(S_1 \sqcap S_2)/R = (S_1/R) \sqcap (S_2/R)$ *and* $Q\backslash(S_1 \sqcap S_2) = (Q\backslash S_1) \sqcap (Q \backslash S_2)$,

6. $S/(R_1 \sqcup R_2) = (S/R_1) \sqcup (S/R_2)$ *and* $(Q_1 \sqcup Q_2)\backslash S = (Q_1\backslash S) \sqcup (Q_2 \backslash S)$.

Lemma 3.4.2. *Let $\mathcal{R}$ be a division allegory. If $Q : A \to B$, $R : B \to C$, $S : A \to C$, $F : D \to A$, and $G : C \to E$ then we have,*

1. $S/\mathbb{I}_C = S$ *and* $\mathbb{I}_A\backslash S = S$,
2. $F;(S/R) \sqsubseteq (F;S)/R$ *and* $(Q\backslash S);G \sqsubseteq Q\backslash(S;G)$,
3. *If F and $G^{\smile}$ are mappings, then in both properties of (2) equality holds,*
4. $S/R \sqsubseteq (S;G)/(R;G)$ *and* $Q\backslash S \sqsubseteq (F;Q)\backslash(F;S)$,
5. *If G and $F^{\smile}$ are total and injective, then in both properties of (4) equality holds.*

The following lemma shows some fundamental properties of symmetric quotients [19].

Lemma 3.4.3. *Let $\mathcal{R}$ be a division allegory. If $Q : A \to B$, $R : A \to C$, $S : A \to D$ are arbitrary relations and $f : D \to B$ is a mapping. Then we have*

1. $f; syQ(Q,R) = syQ(Q; f^{\smile}, R)$,
2. $syQ(Q,R)^{\smile} = syQ(R,Q)$,
3. $syQ(Q,R); syQ(R,S) \sqsubseteq syQ(Q,S)$,

3.5 Heyting Categories

A Heyting category is a division allegory in which every $\mathcal{R}[A,B]$ is a Heyting algebra.

Definition 3.5.1. *A division allegory $\mathcal{R}$ is called Heyting category iff every $\mathcal{R}[A,B]$ is a Heyting algebra. We denote the greatest element by $\overline{\top}_{AB}$.*

The next lemma will state some properties of the greatest element in Heyting categories in [19].

Lemma 3.5.1. *Let $\mathcal{R}$ be a Heyting category with objects A and B. Then we have,*

1. $\overline{\top}^{\smile}_{AB} = \overline{\top}_{BA}$,
2. $\overline{\top}_{AA}; \overline{\top}_{AB} = \overline{\top}_{AB}; \overline{\top}_{BB} = \overline{\top}_{AB}$,
2. $\overline{\top}_{AB} = \overline{\top}_{AB}; \overline{\top}_{BA}; \overline{\top}_{AB}$.

The next lemma summarize some additional properties of relations in Heyting categories [19].

Lemma 3.5.2. *Let $\mathcal{R}$ be a Heyting category with some relations, $Q : A \to B$,$R : B \to C$,$S : A \to D$, and $T : D \to C$. Then we have,*

1. $(Q \sqcap S ; \top_{DB}); R = Q; R \sqcap S ; \top_{DC}$,
2. $Q; (R \sqcap \top_{BD}; T); R = Q; R \sqcap \top_{AD}; T$,
3. $\mathbb{I}_A \sqcap Q; Q^{\smile} = \mathbb{I}_A \sqcap Q; \top_{BA} = \mathbb{I}_A \sqcap \top_{AB}; Q^{\smile}$,
4. *Q is total iff* $Q; \top_{BC} = \top_{AC}$, *for all objects C.*

For partial identities, there are some other properties in a Heyting category. The next lemma summarizes these properties [19].

Lemma 3.5.3. *Let $\mathcal{R}$ be a Heyting category, $R : C \to A$, $U : A \to B$ be relations, and $S : A \to A$ be a partial identity. Then we have,*

1. $S = \mathbb{I}_A \sqcap S ; \top_{AA} = \mathbb{I}_A \sqcap \top_{AA}; S$,
2. $R; S = R \sqcap \top_{CA}; S$ *and* $S ; U = U \sqcap S ; \top_{AB}$.

3.6 Schröder Categories

We are now switching from Heyting algebras to Boolean algebras as the underlying lattice structure of relations.

Definition 3.6.1. *A Heyting category where $\mathcal{R}[A, B]$ is a Boolean algebra, is called Schröder category.*

In the next theorem is the demonstration of the so-called Schröder equivalences. In [15] they are used as a basic axiom for relations. In the presence of the other axioms of a Schröder category, the Schröder equivalences are equivalent to the modular law of allegories.

Theorem 3.6.1. *(Schröder equivalences) Let $\mathcal{R}$ be a Schröder category with relation $Q : A \to B$, $R : B \to C$, and $S : A \to C$. Then we have,*

$$Q; R \sqsubseteq S \iff Q^{\smile}; \bar{S} \sqsubseteq \bar{R} \iff \bar{S} ; R^{\smile} \sqsubseteq \bar{Q}$$

In the presence of complements the residuals can be defined using composition, converse, and complement. The next lemma is shown in [19].

Lemma 3.6.1. *Let $\mathcal{R}$ be a Schröder category with relation $Q : A \to B$, $R : B \to C$, and $S : A \to C$. Then we have,*

1. $Q \backslash S = \overline{Q^{\smile}; \bar{S}}$,
2. $S / R = \overline{\bar{S} ; R^{\smile}}$.

Sometimes the so-called Tarski-rule is needed. It emphasizes that relations are based on a two-valued logic.

Definition 3.6.2. *Let $\mathcal{R}$ be a Schröder category with objects A, B, C, D and a relation $R : A \to B$. The Tarski-ruke is the following axiom:*

$$\top_{CA}; R; \top_{BD} = \top_{CD} \text{ if } R \neq \bot_{AB}$$

We use this rule for proving several lemmas and also for algorithm verification.

3.7 Unit Object

A unit is an object. It is an abstract version of a singleton set, i.e., a set with exactly one element.

Definition 3.7.1. *An object 1 is called a unit if $\top_{11} = \mathbb{I}_1$ and $\top_{A1}$ is total for every object A.*

Lemma 3.7.1. $\top_{A,1}; \top_{1,A} = \top$, *for all objects A.*

If we represent a relation $v : A \to 1$ by a Boolean matrix we obtain a matrix similar to a vector in linear algebra. Such a relation can be seen as a subset of A. For example, if $A = \{1, 2, 3, 4, 5\}$ then v is a vector representing the subset { 1, 3, 5 }.

$$v = \begin{pmatrix} 1 \\ 0 \\ 1 \\ 0 \\ 1 \end{pmatrix}$$

Definition 3.7.2. *A relation $v : A \to 1$ is called a vector.*

An element of A can be represented by a singleton subset of A. Therefore, we obtain the following definition of a point.

Definition 3.7.3. *A point (or element) is a surjective and injective vector.*

3.8 Cardinality of Relations

In mathematics, cardinality is used to measure the number of element in a set. For example, the set $A = \{1, 3, 5\}$ has three element, i.e., its cardinality is 3. There two ways of defining cardinality - one is comparing the sets using bijections and injections, and another way is

using cardinal numbers.

In [6] Kawahara investigates the cardinality for set-theoretic relations. The primary outcome is a formula which is called Dedekind inequality. That is used for calculation with cardinalities of relations, and later as an axiom for an algebraic characterization of a cardinality function in [1]. The algebraic definition of a cardinality function uses the notion of an ordered monoid.

A algebraic structure $(M, +)$ of a set M and an associative operation $+$ is called semigroup. A monoid is a semigroup together with a neutral element 0, i.e., $x + 0 = 0 + x = x$ for all $x \in M$. A monoid in which $+$ is commutative is called commutative monoid.

Definition 3.8.1. *An ordered monoid is a monoid M together with a partial order $\sqsubseteq$ so that $+$ is monotonic, i.e., $v + x \sqsubseteq w + y$ if $v \sqsubseteq w$ and $x \sqsubseteq y$ for all $v, w, x, y \in M$.*

Sometimes it is convenient to have a notation for adding an element multiples times.

Definition 3.8.2. *If M is a monoid, $n \in \mathbb{N}$, and $x \in M$, then we define the multiplication $n \cdot x$ recursivley as*

$$0 \cdot x := 0,$$
$$(n+1) \cdot x := x + n \cdot x.$$

Definition 3.8.3. *A cardinality function $|.|$ is a map assigning to every relation R an element $|R|$ of an ordered monoid such that*

1. *$|R| = 0$ iff $R = \perp\!\!\!\perp$,*
2. *$|R| = |R^{\smile}|$,*
3. *$|R \sqcup S| + |R \sqcap S| = |R| + |S|$,*
4. *If Q is univalent, then $|R \sqcap Q^{\smile}; S| \sqsubseteq |Q; R \sqcap S|$,*
5. *If Q is univalent, then $|Q \sqcap S; R^{\smile}| \sqsubseteq |Q; R \sqcap S|$.*

Some properties of a cardinality function are listed in the next few lemmas. A proof can be found in [1]. If the allegory has a unit, then we define $1 := |\top_{11}|$. Please note that 1 is an element of M.

Lemma 3.8.1. *If $R : A \to B$ is univalent and $S : B \to C$ is a mapping, then $|R; S| = |R|$.*

Lemma 3.8.2. *Let $R : A \to A$ be symmetric and $P, Q : A \to C$ with P injective and Q univalent, then we have $|R \sqcap P; Q^{\smile}| = |R; P \sqcap Q|$.*

Lemma 3.8.3. *If $q : A \to 1$ is a point, then $|q| = 1$.*

Lemma 3.8.4. *Let relations $R, S : A \to B$. Then $R \sqsubseteq S \Rightarrow |R| \sqsubseteq |S|$.*

Lemma 3.8.5. $|\perp\!\!\!\perp| = 0$.

3.9 Direct Product

Elements from two sets X, Y can form pairs which are the elements of the Cartesian product $X \times Y$. We denote a pair by $(x, y) \in X \times Y$ where $x \in X$ and $y \in Y$.

Let assume two sets, one of them students name and another set is the marks they may obtain, e.g., $X = \{Adam, Jack, Kelly\}$ and a $Y = \{80, 75, 70\}$. The direct product of sets is the set of all pairs:

$X \times Y$ = {(Adam,80), (Adam,75), (Adam,70),(Jack,80), (Jack,75), (Jack,70),(Kelly,80), (Kelly,75), (Kelly,70)}

3.9.1 Projections

The projection functions allow retrieving the two components from a pair. The first projection obtaining the first element from a pair is denoted by π and second projection is denoted by ρ.Seen as relation in an allegory they have the following source and target:

$\pi : X \times Y \to X$ and $\rho : X \times Y \to Y$.

3.9.2 Algebraic Properties of the Projection Relations

Algebraists and computer scientists have been investigating products and their property abstractly. A significant part of [15] is dedicated to several aspects of this subject. Mathematicians obtained a set of algebraic rules are always satisfied. In [15], these rules led to the following abstract definition of a direct product.

Definition 3.9.1. *An object $A \times B$ together with two relations $\pi : A \times B \to A$ and $\rho : A \times B \to B$ are said to form a direct product if*

1. $\pi^{\smile}; \pi = \mathbb{I}$,
2. $\rho^{\smile}; \rho = \mathbb{I}$,
3. $\pi; \pi^{\smile} \sqcap \rho; \rho^{\smile} = \mathbb{I}$,
4. $\pi^{\smile}; \rho = \top\!\!\top$.

Note that π, ρ are mappings. For first two conditions require π, ρ to be univalent and surjective and third condition implies that π, ρ are total. The fourth condition implies that for every element in A and B there exists exactly one pair in $A \times B$. There are some interesting constructions defined in [15],

Definition 3.9.2. *Let $R : A \to B$ and $S : A \to Y$ be relations, then the strict fork operation between R and S, denoted by $A \to B \times Y$, is defined by,*

$$(R \circledcirc S) := R;\pi^{\smile} \sqcap S;\rho^{\smile}.$$

Definition 3.9.3. *Let $R : B \to A$ and $S : Y \to A$, then the strict join operation between relation R and S denoted by $B \times Y \to A$, is defined by,*

$$(R \circledcirc S) := \pi;R \sqcap \rho;S.$$

Definition 3.9.4. *Let $R : A \to B$ and $S : X \to Y$, then the Kronecker product between relation R and S denoted by $A \times X \to B \times Y$, is defined as the following,*

$$(R \bigotimes S) := \pi;R;\pi\prime^{\smile} \sqcap \rho;S;\rho\prime^{\smile}.$$

The next three lemmas present some important properties of these operations using the algebraic definition of a direct product [15].

Lemma 3.9.1. *Let $R : A \to B$, $S : X \to Y$ be relations and $\pi : A \times X \to A$, $\rho : A \times X \to X$ and $\pi\prime : B \times Y \to B$, $\rho\prime : B \times Y \to Y$ be projections. Then following properties hold:*

1. $(R \bigotimes S);\pi\prime = \pi;R \sqcap \rho;S;\overline{\top\!\!\top_{YB}} \sqsubseteq \pi;R$,
2. $(R \bigotimes S);\rho\prime = \rho;S \sqcap \rho;R;\overline{\top\!\!\top_{BY}} \sqsubseteq \rho;S$,
3. $(R \bigotimes S);(P \bigotimes Q) \sqsubseteq (R \bigotimes P);(S \bigotimes Q)$.

Lemma 3.9.2. *If $(R \circledcirc S) : A \to B \times Y$ is the strict fork of $R : A \to B$ and $S : A \to Y$. Then*

$$(R \circledcirc S);\pi = R \sqcap S;\top\!\!\top \text{ and } (R \circledcirc S);\rho = S \sqcap R;\top\!\!\top.$$

Analogously, if $(R \circledcirc S) : B \times Y \to A$ is the strict join of $R : B \to A$ and $S : Y \to A$. Then

$$\pi^{\smile};(R \circledcirc S) = R \sqcap \top\!\!\top;S \text{ and } \rho^{\smile};(R \circledcirc S) = S \sqcap \top\!\!\top;R.$$

Lemma 3.9.3. *With the assumptions of Lemma 3.9.1 and Lemma 3.9.2 , the following properties hold:*

1. *If S is total then $(R \bigotimes S);\pi\prime = \pi;R$,*
2. *If R is total then $(R \bigotimes S);\rho\prime = \rho;S$,*
3. *If S is total then $(R \circledcirc S);\pi = R$,*
4. *If R is total then $(R \circledcirc S);\rho = S$,*
5. *If S is surjective then $\pi^{\smile};(R \circledcirc S) = R$,*
6. *If R is surjective then $\rho^{\smile};(R \circledcirc S) = S$,*

3.10 Direct Sum

The direct sum of two sets is a set that combines the elements of the two sets. It is the smallest set which contains the elements from both sets without losing the information from which set an element originated. As a consequence, elements that are in both sets will occur twice in the sum, one copy originating from the first set and one copy originating from the second set. Assume we have two sets, one is for cricket playing countries, and another is for soccer playing countries, i.e., Cricket = {Bangladesh, England, Australia, South Africa} and Soccer = {Argentina, Germany, England}. The direct sum of this two set denoted by Cricket + Soccer is the set. So,

Cricket + Soccer = {Bangladesh$_c$, England$_c$, Australia$_c$, South Africa$_c$, Argentina$_s$, Germany$_s$, England$_s$}

Note that the index of each element in Cricket+Soccer indicates from which set the element originated. All elements from both sets are present here. England plays both Cricket and Soccer. Therefore England appears twice on the set of the direct sum.

3.10.1 Injections

Similar to the projection of a direct sum comes with two functions that inject the elements from each set into the direct sum. Seen as relations of an allegory the injections ι and κ have the following source and target:

$$\iota : X \to X + Y \text{ and } \kappa : Y \to X + Y.$$

3.10.2 Algebraic Properties of Injection Relations

According to [15] we can define direct sum as follows.

Definition 3.10.1. *An object $A + B$ together with two relations $\iota : A \to A + B$ and $\kappa : B \to A + B$ is said to form a direct sum if*

1. $\iota ; \iota^{\smile} = \mathbb{I}$,
2. $\kappa ; \kappa^{\smile} = \mathbb{I}$,
3. $\iota^{\smile} ; \iota \sqcup \kappa^{\smile} ; \kappa = \mathbb{I}$,
4. $\iota ; \kappa^{\smile} = \mathbb{L}$.

Now we state some lemmas according to [15].

Lemma 3.10.1. *Let $R : C \to A$ and $S : C \to B$ be relations, then following properties hold:*

1. $(R; \iota \sqcup S; \kappa); \iota^{\smile} = R$,
2. $(R; \iota \sqcup S; \kappa); \kappa^{\smile} = S$,

Lemma 3.10.2. *Let $R : A \to C$ and $S : B \to C$ be relations, then following properties hold:*

1. $\iota; (\iota^{\smile}; R \sqcup \kappa^{\smile}; S) = R$,
2. $\kappa; (\iota^{\smile}; R \sqcup \kappa^{\smile}; S) = S$,

Lemma 3.10.3. *Let $R : A \to B, S : X \to Y$ be relations, then following properties hold:*

1. $(\iota^{\smile}; R; \iota\prime \sqcup \kappa^{\smile}; R; \kappa\prime); \iota\prime^{\smile} = \iota^{\smile}; R$,
2. $(\iota^{\smile}; R; \iota\prime \sqcup \kappa^{\smile}; R; \kappa\prime); \kappa\prime^{\smile} = \kappa^{\smile}; S$,

3.11 Relational Atoms and Edges

In the set-theoretic model, an atom is a relation that consists single pair. If the relation is the incidence relation of a directed graph, then we can say that an atom is a single edge in the graph. An atom can be characterized abstractly as follows.

Definition 3.11.1. *A relation $R : A \to B$ is a atom, if*

1. $R \neq \mathbb{L}$,
2. $R^{\smile}; \top; R \sqsubseteq \mathbb{I}$,
3. $R; \top; R^{\smile} \sqsubseteq \mathbb{I}$,

Please note that every point is an atom among the vectors [16, Proposition 2.4.5].

In an undirected graph an edge is a connection between two nodes. Seen as relation there is a directed edge from x to y and a directed edges from y to x.

Definition 3.11.2. *An edge e is a relation so that there is an atom a with $e = a \sqcup a^{\smile}$.*

Example 3.11.1. *Let assume a set of nodes $X = \{1, 2, 3, 4\}$. The following relation R can be seen as an undirected graph on X. For example, there is an edge between* 1 *and* 2 *because the two entries in the* 1*-row and* 2*-column resp.* 2*-row and* 1*-column are 1.*

$$R = \begin{array}{c} \begin{matrix} 1 & 2 & 3 & 4 \end{matrix} \\ \begin{pmatrix} 0 & 1 & 1 & 0 \\ 1 & 0 & 1 & 1 \\ 1 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 \end{pmatrix} \end{array} \begin{matrix} 1 \\ 2 \\ 3 \\ 4 \end{matrix} \qquad a = \begin{array}{c} \begin{matrix} 1 & 2 & 3 & 4 \end{matrix} \\ \begin{pmatrix} 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{pmatrix} \end{array} \begin{matrix} 1 \\ 2 \\ 3 \\ 4 \end{matrix} \qquad e = \begin{array}{c} \begin{matrix} 1 & 2 & 3 & 4 \end{matrix} \\ \begin{pmatrix} 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{pmatrix} \end{array} \begin{matrix} 1 \\ 2 \\ 3 \\ 4 \end{matrix}$$

The relation a is an atom contained R. This atom leads to the edge e between 1 *and* 2 *as defined above.*

Chapter 4

Approximation Algorithms

The world is now dealing with massive amount of data, and it is ubiquitous in today's society to make a choice by shifting data. We use computer to make a decision rapidly.. Routing a vehicle, organizing data for efficient retrieval are some examples of the real word problems where we use programming for making a decision. Discrete optimization is the field of computer science where we discuss how to achieve the best solution regarding making a decision. Unfortunately, most of the optimization problem are NP-hard. That means there is no efficient algorithm to find the best solution. We might be able to compute a solution in polynomial time, but the solution may or may not be an optimal one.

We always have to consider two criteria when we develop a software solution to a problem; complexity and correctness. We cannot always have an algorithm which gives an optimal solution and also runs in polynomial time. When we are dealing with an NP-hard optimization problem, we need to relax one of that requirement.

Approximation algorithms are algorithms that compute an approximate solution of the optimization problem at hand. As mentioned before, this kind of algorithm may not compute an optimal solution, but we can determine how close the solution is to an optimal solution. The ratio between the result obtained from the algorithm and an optimal solution is called approximation factor. The approximation factor indicates how close the solution is to an optimal solution.

The correctness of a program is always a concern. There are several ways of testing software such as security testing, unit testing, etc. Verification, i.e., a formal proof, ensures that the program satisfies all conditions that are supposed to be satisfied. We say an approximation algorithm is logically correct if it computes a solution to the problem and satisfies

a given approximation factor.

In this chapter, we discuss several optimization problems. These problems are the vertex cover, hitting set, maximum independent set, and the maximum cut problem.

4.1 Vertex Cover

Finding a minimum vertex cover is a typical optimization problem in computer science. It is an NP-hard optimization problem. However, there is a sufficient approximation algorithm. A vertex cover is a set of vertices or nodes so that every edge (u, v) of the graph satisfy that u or v is in vertex cover. The standard input for this problem will be a graph itself and output will be a vertex cover.

4.1.1 Pseudo Code of Approximation Algorithm for the Vertex Cover Problem

The procedure for solving a vertex cover problem is given below:

(1) In the beginning the result is empty and E is the set of edges of the graph.
(2) Do the following until E is empty
    (3a) First take out an arbitrary edge (u, v) from set E.
    (3b) Add u and v to the result.
    (3c) Remove all edges from E which start or end with u or v.
(3) Return result.

4.1.2 Example

Let us consider a graph with the set of vertices $V = \{1, 2, 3, 4, 5, 6, 7, 8\}$ and the list of edges $E = \{((1, 2), (1, 4), (2, 3), (2, 5), (3, 6), (5, 6), (3, 7), (3, 8)\}$. Figure 4.1 shows a representation of the initial graph.

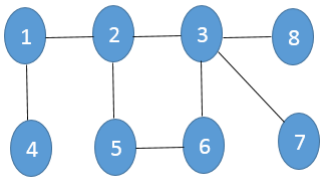


Figure 4.1: Initial Graph

Now if we select an edge (2,3) then we need to remove all edges that start with or end with vertices 2 and 3. So the edges (1,2), (2,5), (2,3), (3,6), (3,7), (3,8) will be removed automatically and the graph will look like Figure 4.2 where the current result is highlighted in red.

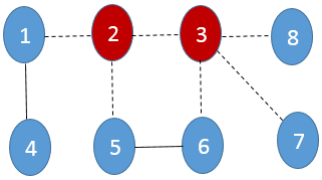


Figure 4.2: Graph after selecting edge (2,3)

Now we need to select the (1,4) and then (5,6) since all other edges have been removed and neither 1,4,5 nor 6 are covered yet. So the output of the vertex cover will be $\{1, 2, 3, 4, 5, 6\}$.

If we select edge (1,2) instead of edge (2,3) in the initial step of the algorithm, then we can remove the edges (1,2), (1,4), (2,3), and (2,5). Figure 4.3 shows that graph.

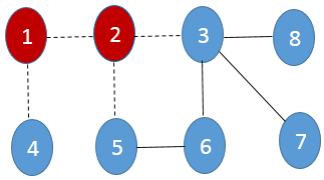


Figure 4.3: Graph after selecting edge (1,2)

Now if we select an edge (3,6) then we can remove all other edges in that graph. So in that case, the output of the vertex cover will be $\{1, 2, 3, 6\}$, which is an optimal solution. Figure 4.4 shows this optimal vertex cover of the graph.

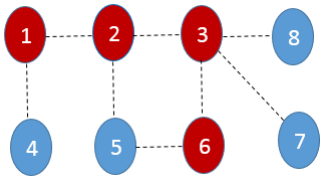


Figure 4.4: Graph after selecting edge (3,6)

4.1.3 Algorithm for the Minimum Vertex Cover Problem

We assume that graph is undirected for this problem. The graph has a non-empty and finite set X of vertices and set of edges called E. Each element in the set E is a pair of two different elements from X. We can represent the graph $G = (X, E)$ by its adjacency relation $R : X \to X$ where xRy means that there is an edge from x to y. Since G is undirected, R is symmetric.

We present the algorithm in two versions. The first version is an imperative version used in [1]. The second version is a functional, and, hence, recursive version. We present the second version in an ML-like syntax. Please note that the actual version in Coq will include additional parameters ensuring termination of the algorithm (see Chapter 7).

For our program, we will take R as an input where we use following formula as a pre-condition, $Pre(R)$ of the program which means R is symmetric.

$$R = R^{\smile}$$

The output of a vertex cover of G is a set of vertices c, i.e., a vector $c : X \to 1$. Here c is a vertex cover of G if and only if $R \sqsubseteq c; \top \sqcup (c; \top)^{\smile}$ where $\top : 1 \to X$. The formula can be read as follows. Every edge of R is included in all potential edges starting in c or ending in c. The cardinality of vertex cover c obtained by the approximation algorithm of Garvil and Yannakakis described above is always less then or equal to twice the cardinality of a minimum vertex cover. Therefore, the conjunction of the following two formulas are the post-condition $Post(R, c)$ of the program.

$$(1)\ R \sqsubseteq c; \top \sqcup (c; \top)^{\smile} \qquad (2)\ \forall d : X \to 1 | R \sqsubseteq d; \top \sqcup (d; \top)^{\smile} \Rightarrow |c| \leqslant 2 \cdot |d|$$

The approximation algorithm of Garvil and Yannakakis is given below:

$c, S := \bot\!\!\!\bot_{X1}, R$
while $S \neq \bot\!\!\!\bot$ *do*
  $e := edge(s)$
  $c, S := c \sqcup e; \top, S \sqcap \overline{e; \top \sqcap \top; e}$

According to algorithm, c is of type $X \to 1$ which is initialized with the empty relation. The relation S is initialized by R and its type can be obtained from R. After that, we extract the first edge and store it in e. Some basic function like union, intersection, complement are used to update the value of c and S. This procedure is repeated until S is empty.

The following lemmas holds the invariant properties(Inv) for the algorithm that described above.

Lemma 4.1.1. *$Pre(R)$ implies $Inv(R, \bot\!\!\!\bot_{x1}, R)$.*

Lemma 4.1.2. *Let $R, S : X \to X$ such that $Inv(R, c, S)$ is satisfied and $S \neq \bot\!\!\!\bot$ then we have $Inv(R, c \sqcup e; \top, S \sqcap \overline{e; \top \sqcap \top; e})$, for all edges $e : X \to X$ with $e \sqsubseteq S$.*

Lemma 4.1.3. *If $R, S : X \to X$ and $C : X \to 1$ satisfy $Inv(R, c, S)$ and $S = \bot\!\!\!\bot$ then $Post(R, c)$ holds.*

The following lemma are stated in [1] indicate the loop termination.

Lemma 4.1.4. *Let $S : X \to X$ with $S \neq \bot\!\!\!\bot$, then $S \sqcap \overline{e; \top \sqcap \top; e} \sqsubset S$, for all edges $e : X \to X$ with $e \sqsubseteq S$.*

In our work, we use a recursive version of the algorithm because Coq does not support looping. Below we present a functional version of that algorithm.

$$
\begin{aligned}
&vertexCover(R) = \\
&\quad if\ R = \bot\!\!\bot\ then\ \ \bot\!\!\bot \\
&\quad else\ let\, e := edge(s)\, in\ (e; \top\!\!\top) \sqcup vertexCover(S \sqcap \overline{e; \top\!\!\top \sqcap \top\!\!\top; e})
\end{aligned}
$$

The following lemma shows that our algorithm is correct.

Lemma 4.1.5. *If $R : X \to X$ satisfies $Pre(R)$, then $c = vertexCover\,R\,p$ satisfies $Post(R, c)$.*

For a proof of the previous lemma we refer to the corresponding proof in Coq.

4.2 Adaption to Hitting Sets

In this section we apply the approximation algorithm of the previous section for the hypergraphs. In a hypergraph an edge can join any number of vertices. This kind of edges are called hyperedges.

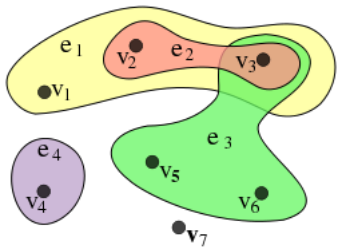


Figure 4.5: Hyper Graph

In Figure 4.5 colors are edges and the nodes within a colored region are incident to that edge. We also notice that node v_3 is a node of the edges e_1, e_2, e_3 and node v_7 is not a node of any edges.

Let assume a hypergraphs $G = (X, E)$ Here X is the non-empty set of vertices and E is the set of hyperedges. According to [16], an incidence relation $I : X \to E$ is used to represent G. Here xIe if and only if $x \in X$, and $e \in E$, and the edge e is incident to the node x.

The cardinality of a maximal hyperedges of G is called the rank of G, which is can be computed relation-algebraically by $max\{|I; p| \mid p : E \to 1\ point\}$. We take the incident

relations I as a input for the relational program [1] and return a vector $c : E \to 1$ as output, which is a vertex cover in the hypergraph. In the context of hypergraphs a vertex cover is called a hitting set. The cardinality of output c of the program will be less then or equal to k-times the cardinality of any hitting set of G where k is the the rank of G. We use the conjunction of following two formulae as the pre-condition $Pre(I,k)$.

$$(1)\ \mathbb{I} \sqsubseteq I^{\smile}; I \qquad\qquad (2)\ k = max\{|I; p| \mid p : E \to 1\ point\}.$$

The first formula requires that I is surjective, which means all hyperedges are a non-empty set of vertices. The second formula states that k is the rank of G.

A short calculation using the incidence relation shows that $c : E \to 1$ is a hitting set of G if and only if $\top\!\!\top = I^{\smile}; c$. The following formula, denoted by $Post(i,k,c)$, is the post-condition of the program. It is the conjunction of c being a hitting set and our desired approximation bound.

$$(1)\ \top\!\!\top = I^{\smile}; c \qquad\qquad (2)\ \forall d : X \to 1, \top\!\!\top = I^{\smile}; d \Rightarrow |c| \leqslant k \cdot |d|.$$

The program which is the adaptation of the vertex cover, to hitting sets and incidence relation is given below.

$c, s := \perp\!\!\!\perp_{E1}, \top\!\!\top$
while $s \neq \perp\!\!\!\perp_{E1}$ *do*
  $p := point(s)$
  $c, s := c \sqcup I; p, s \sqcap \overline{I^{\smile}; I; p}$

The typing $s, p : E \to 1$ can be derived from the type of incidence relation I, initialization of c and the typing rules of the relational operations. Also, we get the type of $\top\!\!\top, \perp\!\!\!\perp : E \to 1$ by the same procedure. In the program, $p = point(s)$ is used instead of $e = edge(s)$ to select a new hyperedge. As a result a new relational-algebraic specification $s \sqcap \overline{I^{\smile}; I; p}$ is used to remove all the hyperedges incident to selected one from the set of hyperedges.

Conjunction of following formulas is the loop invariant $Inv(I,k,c,s)$ of the program above, which is used to prove the correctness of program with respect the to pre-condition $Pre(I,k)$ and the post-condition $Post(I,k,c)$.

$(1)\, Pre(I,k), \qquad (2)\, \bar{s} \sqsubseteq I^{\smile}; c \qquad (3)\, \forall d : X \to 1, \overline{I^{\smile}; I; s} \sqsubseteq I^{\smile}; d \Rightarrow |c| \leqslant k \cdot |d|.$

In [1], the following lemma is shown indicating the termination of the loop.

Lemma 4.2.1. *Let $s : E \to 1$ with $S \neq \bot\!\!\!\bot$, then $s \sqcap \overline{I^{\smile}; I; p} \sqsubset s$, for all edges $p : E \to 1$ with $p \sqsubseteq s$.*

Similar to the corresponding lemmas for the vertex cover problem, the following three lemmas show that the algorithm is correct with respect to $Pre(I,k)$ and $Post(I,k,c)$.

Lemma 4.2.2. *For a relation $I : X \to E$ and $k \in \mathbb{N}$, $Pre(I,k)$ implies $Inv(I,k,\bot\!\!\!\bot_{X1}, \top\!\!\!\top_{X1})$.*

Lemma 4.2.3. *For a relation $I : X \to E, s, c : E \to 1$, and $k \in \mathbb{N}$, $s \neq \bot\!\!\!\bot_{E1}$ and $Inv(I,k,c,s)$, implies $Inv(I,k,c \sqcup I; p, s \sqcap \overline{I^{\smile}; I; p})$ for all points p with $p \sqsubseteq s$.*

Lemma 4.2.4. *$Inv(I,k,c)$ and $s = \bot\!\!\!\bot$ implies $Post(I,k,c)$.*

The following recursive algorithm is the version that we will implement to solve the hitting set problem. The implementation uses a recursive function *hittingSets'* with the additional parameters c and s. The function *hittingSets* calls *hittingSets'* with the initial values $\bot\!\!\!\bot$ and $\top\!\!\!\top$ for c and s, respectively.

hittingSets'(I,c,s) =
if $s = \bot\!\!\!\bot$ then c
else let $p := point(s)$ in hittingSets' I $(c \sqcup I; p)$ $(s \sqcap \overline{I^{\smile}; I; p})$

hittingSets(I) = hittingSets' I $\bot\!\!\!\bot$ $\top\!\!\!\top$.

Here I is the incidence relation which represent a hypergraphs $G = (X,E)$. The following two lemmas indicate that our recursive algorithm is correct with respect to the pre-condition $Pre(I,k)$ and the post-condition $Post(I,k,c)$.

Lemma 4.2.5. *For a relation $I : X \to E$, $k \in \mathbb{N}$ $Inv(I,k,c)$ implies $Inv(I,k,c')$ where $c' =$ hittingSets' I c s.*

Lemma 4.2.6. *For a relation $I : X \to E$ and $k \in \mathbb{N}$ we have that $Pre(I,k)$ implies $Post(I,k,c)$ where $c =$ hittingSets I.*

4.3 Maximum Independent Sets

In graph theory, an independent set is a set of vertices where any two vertices are not connected by an edge, i.e., they are not adjacent. Equivalently there is only one end point in that set for each edge in the graph. Independent sets also called stable sets.

An independent set with a maximum number of vertices in a graph is called maximum independent set. Searching for such a set is called maximum independent set problem. Similar to the previous problems this is an NP-hard optimization problem.

(a) Independent Set (b) Maximum Independent Set

Figure 4.6: Example of Independent Sets

4.3.1 Relational Approximation of Maximum Independent Sets

Let assume a non-empty set of vertices X, for an undirected graph $G = (X, E)$. We represent the graph by an irreflexive and symmetric adjacency relation $R : X \to X$ similar to the vertex cover problem. Therefore, both vertex cover and the maximum independent set problem have almost same preconditions. The approximation bound will be determined by maximum degree k of graph G. So we use a conjunction of these three formulae as pre-condition $Pre(R,k)$ for the program that we use as the solution of the maximum independent set problem.

$$(1)\, R \sqsubseteq \bar{\mathbb{I}} \qquad (2)\, R = R^{\smile} \qquad (3)\, k = max\{|R; p| \mid p : X \to 1\ point\}.$$

A vector $s : X \to 1$ is an independent set if all node adjacent to nodes in s are outside of s, i.e., if $R; s \sqsubseteq \overline{s}$. The cardinality of maximum independent set that we calculate from the program must be less then or equal to $\frac{1}{k+1}$ times the cardinality of any independent sets.

So the approximation bound for the program that we use $\frac{1}{k+1}$. The conjunction of following two formulae will be the post-condition $Post(R, ks)$.

$$(1)\, R; s \sqsubseteq \bar{s} \qquad (2)\, \forall t : X \to 1, R; t \sqsubseteq \bar{t} \Rightarrow |t| \leqslant (k+1) \cdot |s|.$$

We will justify the following program with respect to pre-condition $Pre(R, k)$ and post-condition $Post(R, k, s)$. This program is based on Wei's approximation algorithm described in [18].

$s, v := \perp\!\!\!\perp, \perp\!\!\!\perp_{X1}$
while $v \neq \top\!\!\!\top$ *do*
  $p := point(\bar{v})$
  $s, v := s \sqcup p, v \sqcup p \sqcup R; p$

From the program we can deduce that the type for the variables s, v, p is $X \to 1$. Therefore, the type of $\top\!\!\!\top$should be $X \to 1$ as well. The vector v contains the independent set s computed so far plus all node that are adjacent to s. Therefore, any new node that we would like to add to s must be outside v.

The conjunction of following formulas will be used as the loop invariant $Inv(R, k, s, v)$ when proving the correctness of program with respect to pre-condition $Pre(R, k)$ and the post-condition $Post(R, k, s)$.

$$Pre(R, k), \qquad (4)\, R; s \sqsubseteq \bar{s} \qquad (5)\, R; s \sqcap s = v$$
$$(6)\, \forall t : X \to 1, R; t \sqsubseteq \bar{t} \Rightarrow |t| \leqslant |s| \cdot (k+1).$$

Following lemma showed in [1], indicate the termination of the loop.

Lemma 4.3.1. *Given* $v : X \to 1$ *with* $v \neq \top\!\!\!\top$*and for all points* $p : X \to 1$ *with* $p \sqsubseteq \bar{v}$*, we have* $v \sqsubset v \sqcup p \sqcup R; p$.

The next three lemmas verify that the program is correct with respect to the pre- and post-condition. In particular, Lemma 4.3.1 shows that the precondition establishes the loop invariant, Lemma 4.3.2 shows that the invariant is indeed invariant, and Lemma 4.3.3 shows that the invariant and the complement of the loop condition establishes the post-condition.

Lemma 4.3.2. $Pre(R, k)$ *implies* $Inv(R, k, \perp\!\!\!\perp_{X1}, \perp\!\!\!\perp_{X1})$.

Lemma 4.3.3. *$v \neq \top\!\!\top_{X1}$ and $Inv(R,k,s,v)$ implies $Inv(R,k,s \sqcup p, v \sqcup p \sqcup R; p)$ for all points p with $p \sqsubseteq \bar{v}$.*

Lemma 4.3.4. *$v = \top\!\!\top_{X1}$ and $Inv(R,k,\perp\!\!\!\perp_{X1},\perp\!\!\!\perp_{X1})$ implies $Post(R,k,s)$.*

Similar to hitting set we use the same procedure to declare the recursive algorithm for the maximum independent set problem. After that show the lemma that will prove the correctness of our stated algorithm.

$$
\begin{array}{l}
maxIS'(R,s,v) \;=\; \\
\quad if\ v = \top\!\!\top\ then\ s \\
\quad else\ \ let\, p := point(s)\ in\ maxIS'\ R\ (s \sqcup p)\ (v \sqcup p \sqcup R; p)
\end{array}
$$

$$
maxIS\,(R) \;=\; maxIS'(R)\ \ \perp\!\!\!\perp\ \ \perp\!\!\!\perp.
$$

Lemma 4.3.5. *For a relation $R : X \to X$ and $k \in \mathbb{N}$ $Inv(R,k,s,v)$ implies $Inv(R,k,s',v)$ where $s' = maxIS'\ R\ s\ v$.*

Lemma 4.3.6. *For a relation $R : X \to X$ and $k \in \mathbb{N}$ we have that $Pre(R,k)$ implies $Post(R,k,s)$ where $s = maxIS\ R$.*

Similar to the previous two algorithms we refer to the Coq implementation for a proof of these lemmas.

4.4 Maximum Cut

A cut c in a graph is subset of vertices. The weight of a cut is defined as the number of edges between c and its complement. A maximum cut is a cut with maximal weight, and, hence, the maximum cut problem is the problem of finding a maximum cut. The size of a maximum cut is at least the size of any other cut. As in the previous problems the maximum cut problem is an NP-hard, i.e., there is no polynomial algorithm for computing an optimal solution. The Figure 4.7 shows an example of a maximum cut.

4.4.1 Relational Approximation of Maximum Cuts

Let us assume an undirected loop-free graph $G = (X,E)$. X is the set of non-empty vertices and E is the set of edges between those vertices. As before the graph is given relation-algebraically by symmetric and irreflexive adjacency relation $R : X \to X$. The conjunction

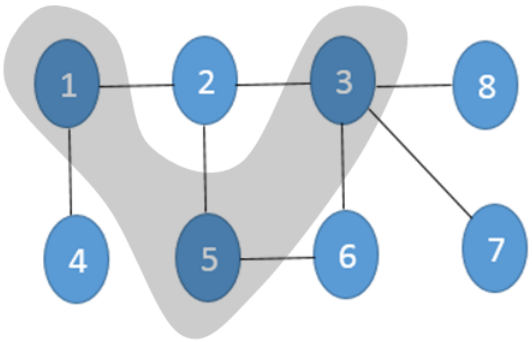


Figure 4.7: A Maximum cut : The vertices {1,5,3}

of following two formulas is the pre-condition $Pre(R)$ of the algorithm.

$$R \subseteq \bar{\mathbb{I}} \qquad R = R^{\smile}$$

We get two disjoint subsets if we apply cut on graph G. With respect to relation R we get a vector $s : X \to 1$ and its complement. In [1], they provide an approximation algorithm for the maximum cut. The relation $R \sqcap (c; \bar{c}^{\smile} \sqcup \bar{c}; c^{\smile})$ restrict the graph to those edges that start in c and end in the complement of c or vice versa. Therefore, its cardinality is the weight of the cut c. The approximation bound of the algorithm is $\frac{1}{2}$. This leads to the following post-condition $Post(R, s)$ where a cut is computed in s:

$$\forall c : X \to 1, |R \sqcap (c; \bar{c}^{\smile} \sqcup \bar{c}; c^{\smile})| \leqslant 2 \cdot |R \sqcap (s; \bar{s}^{\smile} \sqcup \bar{s}; s^{\smile})|.$$

With other words, the post-condition says the weight of the computed cut s is less than or equal to twice the weight of any cut. The following relational program is correct with respect to the pre-condition $Pre(R)$ and post-condition $Post(R, s)$.

$$\begin{array}{l} v, s, t := \overline{\top_{X1}}, \bot\!\!\!\bot, \bot\!\!\!\bot \\ \textit{while } v \neq \bot\!\!\!\bot \textit{ do} \\ \quad p := \textit{point}(v) \\ \quad \textit{if} |R; p \sqcap s| < |R; p \sqcap t| \\ \qquad \textit{then } v, s := v \sqcap \bar{p}, s \sqcup p \\ \qquad \textit{else } \ v, t := v \sqcap \bar{p}, t \sqcup p \end{array}$$

The type of v, s, t and p is $X \to 1$ due to the initialisation of v. The program computes a cut s and its complement with respect to the nodes already visited in t. In each iteration

the number of edges between the current node p and the two set s and t are compared. p is added to the set with fewer edges to p. This approach was mentioned in [12], which is specialization of the approximation algorithm for maximum cut problem published in [14].

The conjunction of following three formulas are considered as loop invariant $Inv(R, v, s, t)$ for the program.

$$(1)\ \ s \sqcap t = \perp\!\!\!\perp \qquad (2)\ \ s \sqcup t = \bar{v} \qquad (3)|R \sqcap (s; s^{\smile} \sqcup t; t^{\smile})| \leqslant |R \sqcap (s; t^{\smile} \sqcup t; s^{\smile})|.$$

The first two formulas state that s and t are a partition of $\overline{v}$, i.e., t is the complement of s with respect to the nodes already visited. Formula (3) says that the number of edges between the set s and t is greater than the number of edges connecting vertices of the set s or t.

The following lemma, stated in [1], shows the termination of the loop.

Lemma 4.4.1. *Given $v : X \to 1$ with $v \neq \perp\!\!\!\perp$, then for all $p : X \to 1$ with $p \sqsubseteq v$, $v \sqcap \bar{p} \sqsubset v$.*

Again, the following lemmas [1] verify that the program is correct with respect to $Pre(R)$ and $Inv(R, v, s, t)$.

Lemma 4.4.2. *If $R : X \to X$ satisfies $Pre(R)$, then $Inv(R, L_{x1}, \perp\!\!\!\perp, \perp\!\!\!\perp)$ holds.*

Lemma 4.4.3. *For all points $p : X \to 1$ with $p \sqsubseteq v$, the following two properties hold if $Pre(R)$ and $Inv(R, v, s, t)$ are satisfied:*
(1) If $|R; p \sqcap s| < |R; p \sqcap t|$, then we have $Inv(R, v \sqcap \bar{p}, s \sqcap p, t)$.
(2) If $|R; p \sqcap t| \leqslant |R; p \sqcap s|$, then we have $Inv(R, v \sqcap \bar{p}, s, t \sqcap p)$.

Lemma 4.4.4. *If $R : X \to X$ and $v, s, t : X \to 1$ such that $v = \perp\!\!\!\perp$ and $Inv(R, v, s, t)$ are satisfied then $Post(R, s)$ holds.*

Finally, we give the recursive version of the algorithm for the maximum cut problem. Similar to the previous two sections, this algorithm is implemented using two functions. We also prove that our defined algorithm is correct.

$$\begin{array}{l} maxCut'(R, v, s, t) \ = \\ \quad if\ v = \perp\!\!\!\perp\ then\ s \\ \quad else\ \ let\ p := point(s)\ in \\ \qquad if\ |R; p \sqcap s| < |R; p \sqcap t|\ then\ maxCut'\ (v \sqcap \bar{p})\ (s \sqcup p)\ t \\ \qquad else\ maxCut'\ (v \sqcap \bar{p})\ s\ (t \sqcup p) \end{array}$$

$$maxCut(R) \;=\; maxCut'\ R\ \top\!\!\top\ \perp\!\!\!\perp\ \perp\!\!\!\perp.$$

Lemma 4.4.5. *For a relation $R : X \to X$ $Inv(R, v, s, t)$ implies $Inv(R, v, s', t)$ where $s' = maxCut'\ R\ v\ s\ t$.*

Lemma 4.4.6. *For a relation $R : X \to X$ we have that $Pre(R)$ implies $Post(R, s)$ where $s = maxCut\ R$.*

Chapter 5

The Coq Proof Assistant

Coq[20] is simultaneously a functional programming language and an interactive proof system. It uses a mathematically high level language called Gallina which is based on the calculus of inductive construction – an expressive formal language. It supports higher-order logic and strongly-typed functional programming. Coq allows to specify theories, their implementation and to prove their correctness. It allows translating certified programs to languages such as Haskel, Objective Caml or Scheme. Coq provides interactive proof methods, decision and semi-decision algorithms and a tactic language as a proof development system. It also provides high-level notations, implicit contents and various other useful tools for the formalization of mathematics or the development of programs.

In this section, we describe some basic feature of Coq that we use frequently. We also give some examples during the discussion. There are lots of other features which are not used and henceforth we did not discuss those features. Details of Coq are available in [20].

5.1 Set, Prop and Type

There are three kinds of types in Coq, and collectively these types are called *sorts*. These three kinds are Set, $Prop$ and $Type$. $Prop$ is the universe of logical propositions. Every theorem is a logical proposition. Set is the universe of specifications and programs. $Type$ is the combined type of Set and $Prop$. $Type$ contain small sets like Boolean, natural numbers, product types and function types over small sets.

The Coq command *Check* is used to obtain the type of a term during an interactive session. Every term has exactly one type.

Check true.
true : bool
Check True.
True : Prop
Check 5.
5 : nat
Check mult.
mult : nat → nat → nat

We need use a period at the end of the statement to terminate that statement.

*Check forall a b, a * b = b * a.*
*forall a b : nat, a * b = b * a : Prop*

As we mention before, every theorem or property P is of type $Prop$, i.e., $P : Prop$. A proof of p of the property P has type P, i.e., $p : P$. The type of a and b in the example above is determined by the type of functions applied to a and b so that we do not have to provide their type in the quantification. However, it is good practice to specify the type of all variables, i.e. instead of $forall\, a\, b$ we should write $forall\, (a\, b : nat)$.

5.2 Proofs and Tactics

Coq provides a formal language to write proofs, almost similar to a programming language. For humans it can be tough to read formal proofs, but for a computer system this is usually not a problem. Human errors can be eliminated by verifying the correctness of a formal proof. This is the major advantage of this approach.

Proofs are done by using tactics and already established facts. In our work, we use induction and substitution rules more frequently. Using the tactic induction does not mean that we are proving the goal actually by induction all the time. Sometimes, especially if the data type involved is not recursive, we use this tactic to distinguish all cases induced by the constructors of the data type. In some cases, it is obvious to prove the goal by induction, especially when lists are involved with the goal.

Often substitution, i.e., replacing a term by another term, is required to move forward towards the goal. Examples include replacing a term using associativity, commutativity,

distributivity and other rules. Let us consider the following proof where we apply some simple tactics to prove the goal.

Lemma Hello_Proof : forall (A B : Prop), A → (A → B) → B.

1 subgoals
————————————————————(1/1)
forall A B : Prop, A → (A → B) → B

In the first line, we declare the lemma. After the announcement of the lemma, Coq generates the goal, shown in second and third line. Now we begin assuming an arbitrary *A* using *intro* the rule.

intro A.

1 subgoals
A : Prop
————————————————————(1/1)
forall B : Prop, A → (A → B) → B

The output above shows the assumption $A : Prop$ above the line and the new goal below the line. Now we use the same rule for assuming an arbitrary *B*.

intro B.

1 subgoals
A : Prop
B : Prop
————————————————————(1/1)
A → (A → B) → B

Again we use the same rule for assuming the left-hand side *A* of the implication, i.e., we assume that *A* is true. Note that this actually means that we assume that we have a proof of *A* due to the type-theoretic interpretation of properties in Coq. In applying the tactic *intro* we also provide a name for that proof explicitly.

intro proof_of_A.

1 subgoals
A : Prop
B : Prop
proof_of_A : A
————————————————————————————(1/1)
(A → B) → B

Again we use the same rule for $A \rightarrow B$.

intro A_implies_B.

1 subgoals
A : Prop
B : Prop
proof_of_A : A
intros A_implies_B : A → B
————————————————————————————(1/1)
B

Now we are done with introducing all the variables and hypotheses. The current goal is just the property B. To prove that goal we can apply the hypothesis $A\_implies\_B$.

apply A_implies_B.

1 subgoals
A : Prop
B : Prop
proof_of_A : A
intros A_implies_B : A → B
————————————————————————————(1/1)
A

Now the goal is A but this is already a hypothesis within our proof named $proof\_of\_A$. We could apply that hypothesis or simply use the tactic *trivial*, which checks whether the goal

is among the hypotheses.

trivial.

No more subgoals.

Finally we have no more goals, which means that our lemma has been shown. We use the tactic *Qed* to finish the proof.

Qed.

Hello_Proof is defined

As lemma $Hello\_Proof$ is defined, we can use that lemma in a different proof similar to a function call. Now we give another example where we prove the commutativity law of $+$ for natural number.

Lemma Plus_Comm : forall x y , x + y = y + x.

1 subgoals
————————————————————————(1/1)
forall x y : nat, x + y = y + x

This time we use the tactic $intros$ which will apply *intro* as often as possible. In our example it will assume two arbitrary elements x, y of type *nat*.

intros.

1 subgoals
x : nat
y : nat
————————————————————————(1/1)
x + y = y + x

Now we proceed by applying induction on x.

induction x.

2 subgoals
y : nat
————————————————————1/2)
0 + y = y + 0
————————————————————(2/2)
S x + y = y + S x

If we apply induction to x, then we get two goals. One is for the base case $x = 0$, and another is for the recursive case where x is the successor of a natural number. Here S denotes the successor function. Proving the first goal is easy. If we apply the tactic *auto*, then system tries to solve the current goal by using a combination of applying hypotheses, introduction and reduction rules.

auto.

1 subgoals
y : nat
IHx : x + y = y + x
————————————————————(1/1)
S x + y = y + S x

We are left with the induction step. The system has automatically added the induction hypothesis IHx while executing the tactic *induction*. Now we use the tactic *simpl* which simplifies an expression by using the definition of the elements and functions in the term.

simpl.

1 subgoals
y : nat
IHx : x + y = y + x
————————————————————(1/1)
S (x + y) = y + S x

To make progress towards the goal, we now want to use the hypothesis IHx. We use the

rewrite tactic for that purpose. The tactic *rewrite* can be used if the property is an equation. The tactic will replace any occurrence of the term on the left-hand side of the property by the right-hand side in the goal. After rewriting IHx we would like to use the tactic *auto* to finish the current goal. We can combine these to steps into one step by using ;.

rewrite IHx;auto.

No more subgoals.

Finally, we use *Qed* again to finish the proof.

Qed.

Plus_Comm is defined

Declaration and summary of that lemma are given, followed by confirmation that lemma has been successfully defined.

Lemma Plus_Comm : forall a b, a + b = b + a.
intros.
induction a.
auto.
simpl.
rewrite IHa;auto.
Qed.

Plus_Comm is defined

Beside these tactics, there are lots of other tactics (see [20]). Every tactic is used for different purpose.

5.3 Classes

Overloading is one of the important concepts of object-oriented programming. This feature allows using the same name for different implementations of an element. In Coq, the concept of overloading can be implemented using type classes.

A type class is a collection of elements declarations. Every instance of the class has to provide an implementation of each declaration. The concept of classes is used widely in several functional programming languages like Haskell and Isabelle. Next, we will show how to declare a class in Coq.

Class Name $(A_1 : T_1)(A_2 : T_2).....(A_N : T_N)$ *:=* {
$F_1 : Q_1$,
$F_2 : Q_2$,
.
.
.
$F_N : Q_N$
}.

After declaring a class followed by a name, we need to provide a type for each component of the class. Note that a class can be parametric, i.e., the class depends on the parameters $A_1, \ldots, A_N$. Now we can declare an instance of that class in the following way:

Instance Name $t_1\, t_2.....t_N$ *:=* {
$F_1 := B_1$,
$F_2 := B_2$,
.
.
.
$F_N := B_N$
}.

A simple example is given below in which the class requires a Boolean valued comparison operation on the type *Ob*. The type *Ob* is a parameter of the class. First, we declare a class and then provide instance for the data type bool of Boolean values.

Class EqualDec (Ob : Type) := {
eqD : Ob $\rightarrow$ *Ob* $\rightarrow$ *bool*
}.

Instance EqualDecBool : EqualDec bool := {
    eqD := fun x y ⇒ if (bool_dec x y) then true else false
}.

Function *bool_dec* takes two Booleans $b1$ and $b2$ as a parameter and returns $\{b1 = b2\} + \{b1 <> b2\}$.

5.4 Functions

The function is the standard feature for almost all programming languages. It is the fundamental principal for a functional programming language. Coq allows defining a function similar to other programming languages. In Coq, functions are normally declared in curried form, i.e., a function f taking two parameters of type A and B returning a value of type C will normally have the type $f : A \rightarrow B \rightarrow C$. Here we give an example below:

Definition inner {a : Type} : (a → bool) → nat → a → nat := fun p n x ⇒ if p x then n+1 else n-1.

If we consider the function above, then we can see that the declaration of any entity starts with keyword $Definition$. After that, we need to provide a name for the entity, which is then followed by a type. In our case, the name is inner. Providing the type is optional. Note that the type of inner should be $inner : forall\,(a : Type) : (a \rightarrow bool) \rightarrow nat \rightarrow a \rightarrow nat$. In this example, we have decided to make the parameter a implicit. This means that we do not have to provide the parameter explicitly when calling inner. The system will try to infer what a is. Besides A the function inner takes three parameters. The first parameter is a predicate on a, i.e., a function that returns a Boolean for every element of type a. The next parameter is a natural number, and the third parameter is of type a. The keyword fun followed by three variables name p, n, x is a lambda abstraction defining a function with parameter names p, n, x of the corresponding type. Alternatively, we could have defined inner as follows avoiding the lambda abstraction syntactically.

Definition inner {a : Type} (p : a → bool) (n : nat) (x : a) : nat := if p x then n+1 else n-1.

Both declarations are equivalent.

5.4.1 Fixpoint

To define a recursive function in Coq we have to use the keyword $Fixpoint$. Coq enforces termination, so that the value of the argument in the recursive call should be decreasing. If we define a function using the keyword $Fixpoint$ this means that the parameter must be smaller in terms of the declaration of the data type of the parameter. For example, if we define a function recursively on *nat*, the recursive case has to be defined for a successor number $S\,x$ and the recursive call has to be on x. Let us consider the following example which also indicates inductive pattern matching.

```
Fixpoint fibonacci (n:nat) : nat :=
    match n with
    | O ⇒ 1
    | (S n1) ⇒
        match n1 with
        | O ⇒ 1
        | (S n2) ⇒ (fibonacci n2) + (fibonacci n1)
        end
    end.
```

A Fibonacci number is a number which is the summation of previous two Fibonacci number. So we need to do pattern matching twice. One on the original parameter of the function and a second on the predecessor of that parameter in the case that it was not zero. For each matching we consider two cases for the natural number - either 0 or successor of some number. In both base cases we return 1 and in the inductive case, we recursively call function $fibonacci$ twice on the smaller elements $n1$ and $n2$. Induction and recursion are used a lot through our work, particularly when lists are involved.

5.5 Infix Operators

It is more user-friendly to use operator symbols instead of function names or properties. In Coq, infix operators can be declared as follows,

```
Infix "+" := plus (at level 50, left associativity).
```

Here + is the left-associative operator for function plus where precedence level is 50. In Coq, ”right associativity” is used after the level declaration if we want right associativity for an operator, i.e., if we want that parsing a sequence of several additions is treated as if brackets are inserted to the right. A lower level indicates a higher precedence of the operator.

In order to declare a postfix operator the keyword *Notation* is used.

Notation ”n !” := (factorial n) (at level 50).

Here *n* is the variable with an appropriate type which is use as a parameter for the function that is mentioned on the right-hand side of the notation declaration.

5.6 Prop vs. bool

According to [20], *Prop* is the universe of a logical Proposition. Properties or reasoning about program constructions is usually done within *Prop*. With other words, the type *Prop* is the type of all logical propositions. Therefore *Prop* has infinitely many elements. Each element of *Prop* is said to be true iff it is provable.

Some properties about *Prop* are not provable without taking any additional assumption.

Lemma isPropEqual : forall (x y : Prop), $x = y \bigvee x <> y$.

This lemma states that, x is either equal to y or not for all proposition x and y.

We cannot apply a case analysis on *Prop* since *Prop* is not defined inductively. In addition some propositions cannot be proved either true or false. The reason is that Coq implements constructive logic in which law of excluded middle does not hold, i.e., $x \bigvee \neg\ x$ cannot be shown for all propositions x.

The type *bool* consists of exactly two elements, the values *true* and $false$. Therefore, case analysis on *bool* is possible.

Lemma isPropBoolean : forall (x y : bool), x = y \/ x <> y.

In our implementation, we handle similar types of problems. So the relationship between *bool* and *Prop* needs to be understood.

5.7 Well-Founded Recursion

In most traditional languages we can call a recursive function without knowing whether the function will terminate or not. One of the significant properties of Coq is termination of every program. This is necessary because of the type-theoretic interpretation. Recall that an element P of type *Prop* is considered true if there is a proof of P. A proof of P is an element of P, i.e., a program of type P. If we allow non-terminating programs, then for every type there is a program with that type, namely the infinite loop. This would imply that all propositions are true. To check completion of all recursive definitions, Coq provides a set of conservative, syntactic criteria which is not sufficient to support natural encodings of a variety of important programming idioms.

In essence, a recursive program will be terminate if there is no infinite chain of nested recursive calls. Coq uses the idea of a well-founded relation to implementing more complex recursions. This technique in Coq is called well-founded recursion. In many cases, we need to provide such a well-founded relation in order to guarantee termination. Please not that the *Fixpoint* construction uses the syntactic subterm relation which is always well-founded.

To define a well-founded relation we need to know about following terms.

Print well_founded.

well_founded = fun (A : Type) (R : A → A → Prop) ⇒ forall a : A, Acc R a

According to the implementation above, we need to show that every element a is accessible in R ($Acc\,R\,a$) in order to verify that R is well-founded. The implementation of *Acc* is as follows.

Print Acc.

Inductive Acc (A : Type) (R : A → A → Prop) (x : A) : Prop := Acc_intro : (forall y : A, R y x → Acc R y) → Acc R x

According to the declaration of *Acc*, an element x is accessible for a relation R if every element less than x according to relation R is also accessible. Since *Acc* is defined inductively this implies that only finite chains are considered. As a consequence only relations for which every chain downwards is finite can be well-founded.

Given a well-founded relation R we can use this relation to define a recursive function. The function Fix of the standard libraries of Coq is used to define such a recursive function.

Check Fix.

```
Fix
     : forall (A : Type) (R : A → A → Prop),
       well_founded R
       forall P : A → Type,
       (forall x : A, (forall y : A, R y x → P y) → P x) →
       forall x : A, P x
```

If we want to call Fix we have to provide a relation R and a proof that R is well-founded. The second parameter is only important if we define a dependently typed function recursively. Since we are not using this feature, we will not go into details. The following line is an encoding of the body of the function. The input x stands for function argument and the second one is the recursive call with any element smaller than x. Using these two parameters we have to compute the result for x. Note that using a recursive call require to provide and element of type Ryx, i.e., a proof that y is less than x with respect to R. Last but not least, $forall\, x : A, P\, x$ is the type of the recursive function defined by applying Fix to the appropriate parameters. Note that this is a dependently typed function. If P does not depend on x, then $forall\, x : A, P\, x$ is simply $A \to P$.

There is another library theorem called Fix_eq which will be used in our implementation. This theorem is used in order to show that a recursively defined function is equal to the function obtained by unfolding the recursion once. This seems to be an obvious fact but needs additional work within Coq since does not support functional extensionality by default. For details we refer to [22,23].

```
Check Fix_eq

Fix_eq
     : forall (A : Type) (R : A → A → Prop) (Rwf : well_founded R)
       (P : A → Type)
       (F : forall x : A, (forall y : A, R y x → P y) → P x),
       (forall (x : A) (f g : forall y : A, R y x → P y),
       (forall (y : A) (p : R y x), f y p = g y p) → F x f = F x g) →
       forall x : A,
       Fix Rwf P F x = F x (fun (y : A) (_ : R y x) ⇒ Fix Rwf P F y)
```

We also need proper induction principle for recursively defined function using a well-founded relation. Coq provides this kind of principle. This important library theorem called *well_founded_induction* which is use to prove the correctness of the recursive program.

```
Check well_founded_induction.

well_founded_induction
     : forall (A : Type) (R : A → A → Prop),
       well_founded R →
       forall P : A → Set,
       (forall x : A, (forall y : A, R y x → P y) → P x) →
       forall a : A, P a
```

More details about well-founded recursion are available in [22].

Chapter 6

Relational Framework

In this chapter, we discuss the implementation of our framework in Coq. We cover both declaration of abstract theory and their implementation as a binary relation. We have separated the proofs of required properties accordingly since we are using a different kind of allegories. To define allegories we first need to define categories, Boolean algebras and other kinds of lattices. In the next chapter, we will use this framework to define the algorithms from the previous chapter and to prove their correctness. We will not go through the details of proving the various theorems in this document. Our complete source code will be found in the online/digital appendix.

6.1 Implementation of Lattices

There two ways of defining lattices. One is according to order theory, and another is an algebraic way. We implement both definitions for our framework.

6.1.1 Order-theoretic Definition of Lattices

Before defining an ordered type in Coq, we need to provide the signature for an order, i.e., a type that has an order relation. We define the signature as follows:

Class OrderSig {*A : Type*} := {
    leq : $A \rightarrow A \rightarrow Prop$
}.

Class *OrderSig* take a parameter A which is a type. The function *leq* represent the actual signature for $(\sqsubseteq)$. We use following infix operator to represent *leq*.

Infix " ⊑" := (leq) (at level 70).

We also define a notation for the reversed order.

Notation "x ⊒ y" := (y ⊑ x) (at level 70, only parsing).

Furthermore, any partial order leads to a corresponding strict order, i.e., to the relationship smaller but not equal.

Definition lt '{S : OrderSig} := fun x y ⇒ (x (⊑) y) ∧ (x <> y).

Again, we introduce proper notation for *lt* and its reversed relation.

Infix "⊏" := (lt) (at level 70).

Notation "x ⊐ y" := (y ⊏ x) (at level 70, only parsing).

The following class implements Definition 2.1.1. Note that the definition below uses the predefined properties *Reflexive* and *Transitive* of Coq.

```
Class Order '(S : OrderSig) := {
    leq_refl :> Reflexive leq;
    leq_trans :> Transitive leq;
    leq_anti : forall(x y : A), x ⊑ y → y ⊑ x → x = y
}.
```

Now we declare following lemmas in Coq regarding orders.

Lemma lt_leq '{A : Order} : forall x y, x ⊏ y → x ⊑ y.

Theorem Equal 'O : Order : forall (x y : A), x = y <-> (forall (z : A), z ⊑ x <-> z ⊑ y).

We need a signature for meet operations. After that, we will denote a meet function by ⊓.

Class MeetSig {A : Type} := {

meet : $A \to A \to A$

}.

Infix "⊓" := (meet) (at level 60, right associativity).

Using the signature for meet operations, we implement an order type with binary meet as follows.

Class MeetOrder '(O : Order) (MS : MeetSig) := {

meet_axiom : forall (x y z : A), $z \sqsubseteq x \sqcap y <\!\!\to z \sqsubseteq x \wedge z \sqsubseteq y$

}.

we define the signature for join operations called $JoinSig$, the notation $\sqcup$, and JoinOrder similarly.

Infix "⊔" := (join) (at level 60, right associativity).

Now we implement order definition of join using join signature.

Class JoinOrder '(O : Order) (JS : JoinSig) := {

join_axiom : forall (x y z : A), $x \sqcup y \sqsubseteq z <\!\!\to x \sqsubseteq z \wedge y \sqsubseteq z$

}.

In [23], Damien Pouse used a Boolean string, implemented by a record named *level* of Booleans, to encode the algebraic hierarchy. His implementation uses a record ops that contains all operations of the hierarchy. This record is parametric in a *level* in order to select the operations that are available in the current structure. For meet and join, this record *level* has two components called *has_cap*, and *has_cup*, respectively. Later, he defines CAP and CUP as concrete levels in which the corresponding bit is set to true. A similar approach is taken for all operations in the algebraic hierarchy.

Finally, we use all the previous declared classes to define order definition of the lattice.

Class LatticeOrder '{O : Order} {MS : MeetSig} (MO : MeetOrder O MS) {JS : JoinSig} (JO : JoinOrder O JS).

These definitions allow us to prove the following two theorems.

Theorem MeetExpand '{MO : MeetOrder} : forall (x y : A), x ⊓ y ⊑ x.

Theorem JoinExpand 'JO : JoinOrder : forall (x y : A), x ⊑ x ⊔ y.

6.1.2 Algebraic Definition of Lattices

Before implementing the algebraic definition of a lattice, we need to define upper semilattices and lower semilattices. We implement both upper and lower semilattices by dividing each into two classes. In the first class we require we associativity and commutativity, and the second class we add the idempotent law. The reason for this separation is that a semilattice requires the idempotency law which will follow from the absorption law in a lattice.

```
Class LSemiLattice' '(MS : MeetSig) := {
    meet_assoc : forall (x y z : A), (x ⊓ y) ⊓ z = x ⊓ (y ⊓ z);
    meet_comm : forall (x y : A), x ⊓ y = y ⊓ x
}.

Class LSemiLattice '(LSL' : LSemiLattice') := {
    meet_idemp : forall (x : A), x ⊓ x = x
}.

Class USemiLattice' '(JS : JoinSig) := {
    join_assoc : forall (x y z : A), (x ⊔ y) ⊔ z = x ⊔ (y ⊔ z);
    join_comm : forall (x y : A), x ⊔ y = y ⊔ x
}.

Class USemiLattice '(USL' : USemiLattice') := {
    join_idemp : forall(x : A), x ⊔ x = x
}.
```

Now we can implement the algebraic definition of a lattice using the class *USemiLattice*, and *LSemiLattice*.

Class Lattice {A : Type} {MS : MeetSig (A:=A)} (LSL' : LSemiLattice' MS) {JS : JoinSig

(A:=A)} (USL' : USemiLattice' JS) := {
meet_absorp : forall(x y : A), x ⊓ (x ⊔ y) = x;
join_absorp : forall(x y : A), x ⊔ (x ⊓ y) = x
}.

Similar to the record *ops* Damien Pous [23] defines a record *laws* that contains all axioms of the algebraic hierarchy. Again, a parameter of type *level* is used to select the appropriate axioms.

Next, we declare a theorem where we prove that the order induced by either the meet operation or the join operation are equivalent.

Theorem OrderEquivDefs '{L : Lattice} : forall (x y : A), x ⊓ y = x <-> x ⊔ y = y.

Now we provide that instances where we show that every lattice is a lower semilattice and also an upper semilattice. As a part of these proofs we verify that lattices are indeed imdepotent as already mentioned above.

Instance LatticeIsLSemiLattice '(L : Lattice) : LSemiLattice LSL'.

Instance LatticeIsUSemiLattice '(L : Lattice) : USemiLattice USL'.

6.1.3 Equivalence of the two Definitions

In the first part, we prove that the algebraic definition of a lattice is equivalent to order-theoretic definition of the lattice. For this purpose, we need to provide an instance of $OrderSig$.

Instance MeetSigToOrderSig '(MS : MeetSig) : OrderSig := {
leq := fun (x y : A) ⇒ x ⊓ y = x
}.

Now we provide two instances where we show that every lower semilatice is an order, in particular, an order with a meet operation.

Instance LSemiLatticeToOrder '(LSL : LSemiLattice) : Order (MeetSigToOrderSig MS).

Instance LSemiLatticeToMeetOrder '(LSL : LSemiLattice) : MeetOrder (LSemiLatticeToOrder LSL) MS.

Using these two instances we can verify that a lattice is also an order with a meet operation.

Instance LatticeToOrder '(L : Lattice) : Order (MeetSigToOrderSig MS) := LSemiLatticeToOrder (LatticeIsLSemiLattice L).

Instance LatticeToMeetOrder '(L : Lattice) : MeetOrder (LatticeToOrder L) MS.

We can easily prove that a lattice is also an order with a join operation using Lemma OrderEquivDefs.

Instance LatticeToJoinOrder '(L : Lattice) : JoinOrder (LatticeToOrder L) JS.

Finally, we can show that every algebraically defined lattice is also a order-theoretic lattice.

Instance LatticeToLatticeOrder '(L : Lattice) : LatticeOrder (LatticeToMeetOrder L) (LatticeToJoinOrder L).

In the second part, we want to verify the opposite implication, i.e., that every order-theoretically defined lattice satisfies the algebraic laws. To show this, we need to establish that order with a meet operation is a lower semilattice, similarly, that every order with a join operation is an upper semilattice.

Definition MeetOrderToLSemiLattice' '(MO : MeetOrder) : LSemiLattice' MS.

Definition MeetOrderToLSemiLattice '(MO : MeetOrder) : LSemiLattice (MeetOrderToLSemiLattice' MO).

Definition JoinOrderToUSemiLattice' '(JO : JoinOrder) : USemiLattice' JS.

Definition JoinOrderToUSemiLattice '(JO : JoinOrder) : USemiLattice (JoinOrderToUSemiLattice' JO).

Now we can easily show our main theorem of this section.

Definition LatticeOrderToLattice '(LO : LatticeOrder) : Lattice (MeetOrderToLSemiLattice' MO)(JoinOrderToUSemiLattice' JO).

Following auxiliary lemma helpful for proving other several lemmas. We can prove that lemma by using axiom meet_axiom and lemma MeetExpand.

Lemma LSLLemma '{LSL : LSemiLattice} : forall (x y : A), $x \sqsubseteq y \rightarrow x \sqcap y = x$.

6.1.4 Distributive Lattices

First we declare the distributivity property for two arbitrary functions.

Class Distributive {A : Type} (f g : $A \rightarrow A \rightarrow A$) :=
distr : forall (x y z : A), f x (g y z) = g (f x y) (f x z).

The next two theorems show that one inclusion of the distributivity laws is always satisfied in a lattice. This is usually called sub-distributivity.

Theorem SubDistrMeet '{L : Lattice} : forall (x y z : A), $x \sqcap y \sqcup x \sqcap z \sqsubseteq x \sqcap (y \sqcup z)$.

Theorem SubDistrJoin '{L : Lattice} : forall (x y z : A), $x \sqcup (y \sqcap z) \sqsubseteq (x \sqcup y) \sqcap (x \sqcup z)$.

We prove an important theorem where we show that the two distributivity laws are equivalent in every lattice.

Theorem EquivDistrLaws '{L : Lattice} : Distributive meet join <-> Distributive join meet.

Before defining a distributive lattice, we define lattices with only one of the two distributive laws. A distributive lattice will require both laws.

Class MDistrLattice '(L : Lattice) :=
mdistr : Distributive meet join.

Class JDistrLattice '(L : Lattice) :=
jdistr : Distributive join meet.

Class DistrLattice '(L : Lattice) := {
meet_distr : Distributive meet join;
join_distr : Distributive join meet
}.

We already know from Theorem *EquivDistrLaws* that one of the distributivity laws would be sufficient. However, we have added both laws to a lattice for convenience. With the following instance declarations it will be sufficient to only prove one of the laws when creating an actual instance of a distributive lattice.

Instance ConstrMDistrLattice '(ML : MDistrLattice) : DistrLattice L.

Instance ConstrJDistrLattice '(JL : JDistrLattice) : DistrLattice L.

Now we prove a valuable property of distributive lattices that we turns out to be useful in proving other properties.

Theorem UniqueCompl '{*L : DistrLattice*} *: forall (x y a : A),* $a \sqcap x = a \sqcap y \vee a \sqcup x = a \sqcup y \rightarrow x = y$.

6.1.5 Declaration of Bounded Lattice

Recall that a bounded lattice is a lattice with a least and greatest element. In our implementation the least element is denoted by *Zero* and the greatest element by *One*. For our convenience, we first define a lattice with a least element and show some related theorems. Then we declare a lattice with a greatest element.

First, we need to specify a signature for a least element.

Class LESig {*A : Type*} := {
Zero : A
}.

The following class implements exactly Definition 2.2.3 restricted to the least element.

Class LELattice '(L : Lattice) (LES : LESig) := {
    le_axiom : forall(x : A), x ⊔ Zero = x
}.

Now we prove the following two important theorems.

Theorem LEProp '{LEL : LELattice} : forall (x : A), Zero ⊑ x.

Theorem LEZeroProp '{LEL : LELattice} : forall (x : A), x ⊓ Zero = Zero.

The case of a greatest element is handled analogously.

Class GESig {A : Type} := {
    One : A
}.

Class GELattice '(L : Lattice) (GES : GESig) := {
    ge_axiom : forall(x : A), x ⊓ One = x
}.

Theorem GEProp '{GEL : GELattice} : forall (x : A), x ⊑ One.

Theorem GEOneProp '{GEL : GELattice} : forall (x : A), x ⊔ One = One.

A bounded lattice can be defined by declarations above.

Class BoundedLattice '{L : Lattice} {LES : LESig} (LEL : LELattice L LES) {GES : GESig} (GEL : GELattice L GES).

We can also define bounded distributive lattices by combining the declarations of a distributive lattice and the definitions above.

Class BoundedDistrLattice '{L : Lattice} (DL : DistrLattice L) {LES : LESig} {LEL : LELat-

tice L LES} {GES : GESig} {GEL : GELattice L GES} (BL : BoundedLattice (L:=L) LEL GEL).

6.1.6 Heyting algbera

As before we are going to define a signature for the implication operation of a Heyting algbera. We use the notation ~> to refer to this operation.

Class PCSig {A : Type} := {
    implies : A → A → A
}.

Infix "~>" := (implies) (at level 65).

Now, we define a Heyting algbera as follows.

Class PCLattice '(L : Lattice) (PCS : PCSig) := {
    pc_axiom : forall (x y z : A), z ⊑ x ~> y <-> x ⊓ z ⊑ y
}.

Now we declare an instance where we prove that every Heyting algebra is a distributive lattice.

Instance PCLatticeToDistrLattice '(pl : PCLattice) : DistrLattice L.

In addition, every Heyting algebra has a greatest element which we formalized by the following instance declarations.

Instance PCLatticeToGESig '(PCL : PCLattice) (a : A) : GESig := {
    One := a ~> a
}.

Instance PCLatticeToGELattice '(PCL : PCLattice) (a : A) : GELattice L (PCLatticeToGESig PCL a).

Now we combine the results above for convenience.

Class PCLELattice ‘{L : Lattice} {PCS : PCSig} (PCL : PCLattice L PCS) {LES : LESig} (LEL : LELattice L LES).

Instance PCLELatticeToGESig ‘(PCLE : PCLELattice) : GESig := PCLatticeToGESig PCL Zero.

Instance PCLELatticeToGELattice ‘(PCLE : PCLELattice) : GELattice L (PCLELattice-ToGESig PCLE) := PCLatticeToGELattice PCL Zero.

Instance PCLELatticeToBoundedLattice ‘(PCLE : PCLELattice) : BoundedLattice LEL (PCLELatticeToGELattice PCLE).

Instance PCLELatticeToBoundedDistrLattice ‘(PCLE : PCLELattice) : BoundedDistrLattice (PCLatticeToDistrLattice PCL) (PCLELatticeToBoundedLattice PCLE).

The implication operation gives rise to a pseudo-complement defined by x~>Zero.

Definition complement ‘{PCL : PCLELattice} := fun x : A ⇒ x ~> Zero.

Notation ”x ~” := (complement x) (at level 50, left associativity).

Now we prove several important theorems.

Theorem ComplementAnd ‘{PCL : PCLELattice} : forall (x : A), x ⊓ x ~= Zero.

Theorem DoubleComplementRel ‘{PCL : PCLELattice} : forall (x : A), x ⊑ x ~~.

Theorem complement_more ‘{PCL : PCLELattice} : forall (x y : A), y ⊑ x → x ~⊑ y ~.

Theorem DoubleComplClo1 ‘{PCL : PCLELattice} : forall (x : A), x ~= x ~~~.

Theorem DoubleComplClo2 ‘{PCL : PCLELattice} : forall (x y : A), (x ⊔ y) ~= x ~⊓ y ~.

Theorem DoubleComplClo2a ‘{PCL : PCLELattice} : forall (x y : A), x ~⊔ y ~⊑ (x ⊓ y) ~.

Theorem DoubleComplClo3 '{PCL : PCLELattice} : forall (x y : A), (x ⊓ y) ~~= x ~~⊓ y ~~.

6.1.7 Boolean Algebras

We declare a class called *BooleanAlgebra* which takes a Heyting algebra as parameter. In this declaration we follow exactly Definition 2.4.1.

Class BooleanAlgebra '(PCL : PCLELattice) := {
ba_axiom : forall (x : A), x ⊔ x ~= One
}.

We declare and prove following important theorems.

Theorem EqualityBooleanAlgebra '{BA : BooleanAlgebra} : forall (x : A), x = x ~~.

Theorem DoubleComplClob '{BA : BooleanAlgebra} : forall (x y : A), x ~⊔ y ~= (x ⊓ y) ~.

Lemma neg_join '{BA : BooleanAlgebra} : forall x y, x ~⊑ y <-> x ⊔ y = One.

Lemma convertMeetComplement '{BA : BooleanAlgebra} : forall a b c, a ⊓ b ~⊑ c → a ⊑ b ⊔ c.

Lemma convertUTA '{BA : BooleanAlgebra} : forall a Q R, a ⊑ Q ⊔ R <-> a ⊓ Q ~⊑ R.

Lemma LTOA '{BA : BooleanAlgebra} : forall a, (forall R, R ⊑ a → R = Zero ∨ R = a) → forall Q R, a ⊑ Q ∨ a ⊑ R <-> a ⊑ Q ⊔ R.

Lemma NotZero '{BA : BooleanAlgebra} : Zero ~= One.

Theorem complement_more_lt '{BA : BooleanAlgebra}: forall (x y : A), (x ⊏ y) <-> (y ~⊑ x ~).

Theorem *EqualityBooleanAlgebra_GR '{BA : BooleanAlgebra} : forall (x : A), x ~~= x.*

6.1.8 Binary Relation

In this section, we define the type of set-theoretic relations and provide implementation of the various lattice structures on that type. A relation is implemented as a characteristic function, i.e., it is a function taken parameters from two types *a* and *b* returning a Boolean. We have chosen the type *bool* instead of *Prop* because we will be using relations in our algorithms, and, hence, we need that it is decidable whether a pair is in the relation or not.

Definition Rel a b := a → b → bool.

In Chapter 2, we have already defined the lattice operations on set-theoretic relations. We will follow these definitions closely. Please note that && and || are Coq functions implementing *and* and *or* on Booleans. These will be used in order to define meet and join on relations.

Definition Meet_Rel {a b : Type} : Rel a b → Rel a b → Rel a b := fun (r s : Rel a b) (x : a) (y : b) ⇒ r x y && s x y.

Definition Join_Rel {a b : Type} : Rel a b → Rel a b → Rel a b := fun (r s : Rel a b) (x : a) (y : b) ⇒ r x y || s x y.

Definition Zero_Rel {a b : Type} : Rel a b := fun (x : a) (y : b) ⇒ false.

Definition One_Rel {a b : Type} : Rel a b := fun (x : a) (y : b) ⇒ true.

For negation, Coq provides an operator called *negb*. We can define our implication relation as follows.

Definition Implies_Rel {a b : Type} : Rel a b → Rel a b → Rel a b := fun (r s : Rel a b) (x : a) (y : b) ⇒ (negb (r x y)) || s x y.

Now we can provide an instance for every signature that we need when we instantiate each of the lattice classes.

```
Instance MyRelMeetSig (a b : Type) : MeetSig (A := Rel a b) := {
    meet := Meet_Rel
}.
```

```
Instance MyRelJoinSig (a b : Type) : JoinSig (A := Rel a b) := {
    join := Join_Rel
}.

Instance MyRelLESig (a b : Type) : LESig (A := Rel a b) := {
    Zero := Zero_Rel
}.

Instance MyRelGESig (a b : Type) : GESig (A := Rel a b) := {
    One := One_Rel
}.

Instance MyRelPCSig (a b : Type) : PCSig (A := Rel a b) := {
    implies := Implies_Rel
}.
```

Following two lemmas are used in the following instance declarations. To prove the first lemma, we use the Lemma *function_extensionality* (module *FunctionalExtensionality*). More information about this module can be found in [21].

```
Lemma Rel_Ext (a b : Type) : forall(f g : Rel a b), f = g <-> (forall x y, f x y = g x y).

Lemma implb_lem : forall (x y z : bool), z && (negb x || y) = z <-> x && z && y = x && z.
```

Finally, we provide an instance for every lattice structure based on binary relations.

```
Instance MyRelLSemiLattice' (a b : Type) : LSemiLattice' (MyRelMeetSig a b).

Instance MyRelLSemiLattice (a b : Type) : LSemiLattice (MyRelLSemiLattice' a b).

Instance MyRelUSemiLattice' (a b : Type) : USemiLattice' (MyRelJoinSig a b).

Instance MyRelLattice (a b : Type) : Lattice (MyRelLSemiLattice' a b) (MyRelUSemiLat-
tice' a b).
```

Instance MyRelDistrLattice (a b : Type) : DistrLattice (MyRelLattice a b).

Instance MyRelLELattice (a b : Type) : LELattice (MyRelLattice a b) (MyRelLESig a b).

Instance MyRelGELattice (a b : Type) : GELattice (MyRelLattice a b) (MyRelGESig a b).

Instance MyRelPCLELattice (a b : Type) : PCLELattice (MyRelPCLattice a b) (MyRelLE-Lattice a b).

Instance MyRelBooleanAlgebra (a b : Type) : BooleanAlgebra (MyRelPCLELattice a b).

6.2 Categories and Allegories

In this section, we focus on implementing categories and various allegories. During the declaration, we maintain the hierarchy of allegories.

6.2.1 Categories

We need to provide a signature for composition and the identity as we need those during the declaration of a category. Please note that the signature will depend on two parameters. The first parameter is the type of the objects of the category. The second parameter provides the type of morphisms between to given objects.

Class CategorySig {Obj : Type} (Mor : Obj → Obj → Type) := {
    comp : forall {a b c : Obj}, Mor a b → Mor b c → Mor a c;
    ident : forall {a: Obj}, Mor a a
}.

We use notation id to represent ident and ∘ to represent composition.

Notation "id" := (ident).

Infix "∘" := (comp) (at level 55, right associativity).

The following implements Definition 3.1.1.

Class Category '(CS : CategorySig) := {
assoc: forall(a b c d : Obj) (f: Mor a b) (g: Mor b c) (h: Mor c d),
(f ∘ g) ∘ h = f ∘ (g ∘ h);
idl_law : forall (a b : Obj) (f : Mor a b), id ∘ f = f;
idr_law : forall (a b : Obj) (f : Mor a b), f ∘ id = f
}.

6.2.2 Allegories

According to Definition 3.2.1 an operation called converse is part of an allegory so that we need to define a signature and appropriate notation for this operation.

Class AllegorySig '(C : Category) := {
converse : forall {a b: Obj}, Mor a b → Mor b a
}.

Notation "˘ " := (converse x) (at level 50, left associativity).

For implementing the definition of allegories from Chapter 3, we need to provide an implementation of the a category, the signature for an allegory, and the signature and of a lower semilattice structure for each collection of morphisms.

Class Allegory '(C : Category)
(MS : forall a b : Obj, MeetSig (A := Mor a b))
(LSL' : forall a b : Obj, LSemiLattice' (MS a b))
(LSL : forall a b : Obj, LSemiLattice (LSL' a b))
(AS : AllegorySig C) := {
axiom_2a: forall (a b : Obj) (Q R : Mor a b), Q ⊑ R → Q ˘ ⊑ R ˘;
axiom_2b: forall(a b c : Obj) (Q : Mor a b) (S : Mor b c), (Q ∘ S) ˘ = S ˘ ∘ Q ˘;
axiom_2c: forall(a b : Obj) (Q : Mor a b), Q ˘ ˘ = Q;
axiom_3: forall(a b c : Obj) (Q : Mor a b) (R : Mor b c) (S : Mor b c),
Q ∘ (R ⊓ S) ⊑ Q ∘ R ⊓ Q ∘ S;
axiom_4: forall(a b c : Obj) (Q : Mor a b) (R : Mor b c) (S : Mor a c),
Q ∘ R ⊓ S ⊑ Q ∘ (R ⊓ Q ˘ ∘ S)
}.

We have shown Lemma 3.2.1 in Coq. In addition we have the following.

Theorem Monotony '{A : Allegory} : forall{a b c : Obj} (P Q : Mor a b) (R S : Mor b c), (P ⊓ Q) ∘ (R ⊓ S) ⊑ P ∘ R ⊓ Q ∘ S.

Our next task is to implement Definition 3.2.2.

Definition univalent '{A : Allegory} {a b : Obj} (Q : Mor a b) := Q˘ ∘ Q ⊑ id.

Definition total '{A : Allegory} {a b : Obj} (Q : Mor a b) := id ⊑ Q ∘ Q˘.

Definition map_Rel '{A : Allegory} {a b : Obj} (Q : Mor a b) := (univalent Q) ∧ (total Q).

Definition injective '{A : Allegory} {a b : Obj} (Q : Mor a b) := univalent (Q˘).

Definition surjective '{A : Allegory} {a b : Obj} (Q : Mor a b) := total(Q˘).

Definition bijective '{A : Allegory} {a b : Obj} (Q : Mor a b) := map_Rel (Q˘).

Definition bijection '{A : Allegory} {a b : Obj} (Q : Mor a b) := bijective (Q˘).

Definition symmetric '{A : Allegory} {a : Obj} (Q : Mor a a) := Q˘ = Q.

Using those definitions, we implemented Lemma 3.2.2 and 3.2.3. In addition, the following Lemmas show that the dual properties of Lemma 3.2.2 also holds.

Theorem DualUnivalent1 '{A : Allegory} : forall (a b c : Obj) (Q: Mor b c) (R S: Mor a b), injective Q → (R ⊓ S) ∘ Q = R ∘ Q ⊓ S ∘ Q.

Theorem DualUnivalent2 '{A : Allegory} : forall (a b c : Obj) (Q: Mor b a) (T: Mor a c) (U: Mor b c), injective Q → Q ∘ T ⊓ U = Q ∘ (T ⊓ Q ˘ ∘ U).

Now we implement partial identities according to Definition 3.2.3 and also prove Lemma 3.2.4 that related to partial identities.

Definition PartialIdentities '{A : Allegory} {a : Obj} (R : Mor a a) := R ⊑ id.

In Coq, we can use term coercions when one class resides in another class. In our implementation, we first prove that categories reside in allegories and then we declare the corresponding coercion.

The advantage of coercion is that Coq will use this information automatically while typing expressions.

Instance AllegoryToCategory '(A : Allegory) : Category _.

Coercion AllegoryToCategory : Allegory >-> Category.

6.2.3 The Category and Allegory of Binary Relations

In order to implement composition of two set-theoretic relations we need that the existential quantifier in Definition 2.5.2 ranges over finitely many elements. This implies that we have to restrict ourselves to relations between finite types. A type can be made finite by providing a list of its elements and requiring a proof that all elements are actually included in this list. In addition, we will need to compare elements of each type. For example, this is necessary to define the identity relation. Last but not least, we want that every type is not empty. All these requirements are summarized in the class $FNTDType$ of finite, non-empty types with a decidable equality.

```
Class FNTDType := {
    A : Type;
    elements : list A;
    finite_pr : forall(x : A), In x elements;
    non_empty_pr : elements <> nil;
    Deq : forall x y : A, x = y + x <> y;
    CDeq x y := if Deq x y then true else false
}.
```

The component *Deq* requires a proof that equality on the type A is decidable. This proof is converted by $CDeq$ into a Boolean valued function that compares two elements of type A.

We prove two individual lemmas, which are useful in the rest of the implementation.

Theorem CDeq_true {A : FNTDType}: forall (x y : A), CDeq x y = true <-> x = y.

Theorem CDeq_false {A : FNTDType}: forall (x y : A), CDeq x y = false <-> x <> y.

Now we define the identity relation and the operations of composition, converse and complement. All the definitions are base on the Boolean relation. We get the exactly same result that describes in chapter 2 using following definitions.

Definition ID_Rel (a : FNTDType): Rel a a := fun (x y : A) ⇒ CDeq x y.

Definition Comp_Rel (a b c : FNTDType): Rel a b → Rel b c → Rel a c := fun Q R x z ⇒ existsb (fun y ⇒ (Q x y) && (R y z)) elements.

Definition Converse_Rel (a b : FNTDType): Rel a b → Rel b a := fun R x y ⇒ (R y x).

Definition Complement_Rel (a b : FNTDType) : Rel a b → Rel a b := fun (r : Rel a b) (x : a) (y : b) ⇒ negb (r x y).

Now we provide an instance for both the signature of categories and allegories. In both cases we have to prove that all axioms of categories respectively allegories are satisfied.

```
Instance MyRelCategorySig : CategorySig (Obj := FNTDType) Rel:= {
    comp := Comp_Rel;
    ident := ID_Rel
}.

Instance MyRelCategory : Category (MyRelCategorySig).

Instance MyRelAllegorySig : AllegorySig MyRelCategory := {
converse := Converse_Rel
}.

Instance MyRelAllegory : Allegory MyRelCategory MyRelMeetSig MyRelLSemiLattice'
```

MyRelLSemiLattice MyRelAllegorySig.

6.3 Implementation of Distributive Allegories

According to the definition of distributive allegories in Chapter 3, we need the join operation and a least element. To fulfil those condition, we need to use the classes of lattices with a least element.

A distributive allegory is an allegory so that the corresponding class uses a parameter of type *Allegory*. The implementation of Definition 3.3.1 is given below.

```
Class DistributiveAllegory '(A : Allegory)
        (JS : forall a b : Obj, JoinSig (A := Mor a b))
        (USL' : forall a b : Obj, USemiLattice' (JS a b))
        (USL : forall a b : Obj, USemiLattice (USL' a b))
        (L : forall a b : Obj, Lattice (LSL' a b) (USL' a b))
        (DL : forall a b : Obj, DistrLattice (L a b))
        (LES : forall a b : Obj, LESig (A := Mor a b))
        (LEL : forall a b : Obj, LELattice (L a b) (LES a b)) := {
      axiom_LE: forall (a b c : Obj) (Q : Mor a b),
        (Q ∘ (Zero : (Mor b c))) = (Zero : (Mor a c));
      axiom_Dstr: forall (a b c : Obj) (Q : Mor a b) (R S : Mor b c),
        Q ∘ (R ⊓ S) = Q ∘ R ⊔ Q ∘ S
}.
```

Now we provide an implementation of Lemma 3.3.1. These properties are important because we use those characteristics several times to prove other lemmas.

Since the class *Allegory* resides on Class *DistributiveAllegory*, we provide he corresponding coercion.

```
Instance DistributiveAllegoryToAllegory '(DisA : DistributiveAllegory) : Allegory _ _ _ _ _.

Coercion DistributiveAllegoryToAllegory : DistributiveAllegory >-> Allegory.
```

6.3.1 The Distributive Allegory of Binary Relations

We have already implemented all structures required to make our version of set-theoretic relations an instance of the class *DistributiveAllegory*. In the declaration below it remains to verify the additional axioms.

Instance MyRelDistributiveAllegory : DistributiveAllegory MyRelAllegory MyRelJoinSig MyRelUSemiLattice' MyRelUSemiLattice MyRelLattice MyRelDistrLattice MyRelLESig MyRelLELattice.

6.4 Implementation of Division Allegories

According to hierarchy, our next step is to implement division allegories. In the definition of division allegories in Chapter 3, an additional operation called left residual is used. So we need to declare the signature for this operation.

Class DivisionAllegorySig '(DA : DistributiveAllegory) := {
leftResidual : forall {a b c : Obj}, Mor a c → Mor b c → Mor a b
}.

Infix "//" := (leftResidual) (at level 55, right associativity).

Now we are ready to implement Definition 3.4.1. Obviously, the definition of the class *DivisionAllegory* uses a parameter of type *DivisionAllegorySig*.

Class DivisionAllegory '(DAS : DivisionAllegorySig) := {
axiom_Division: forall (a b c : Obj) (Q : Mor a b) (R : Mor b c) (S : Mor a c),
Q ∘ R ⊑ S <-> Q ⊑ (S // R)
}.

In Chapter 3, we have discussed the right residual and the symmetric. We define these two operations in Coq accordingly.

Definition rightResidual '{DA : DivisionAllegory} := fun {a b c : Obj} (Q : Mor a b) (S : Mor a c) ⇒ (S˘ // Q˘)˘ .

Infix "\\\" := (rightResidual) (at level 55, right associativity).

Definition syQ '{DA : DivisionAllegory} := fun {a b c : Obj} (Q : Mor a b) (R : Mor a c) ⇒ (Q \\\ R) ⊓ (Q ˘ /// R ˘).

The next step is to prove Lemma 3.4.1, 3.4.2 and 3.4.3 in Coq. We have omitted this here. The code can be found in the library. Last but not least, we establish the coercion between division and distributive division allegory and distributive allegory.

Instance DivisionAllegoryToDistributiveAllegory '(DA : DivisionAllegory) : DistributiveAllegory _ _ _ _ _ _ _ _.

Coercion DivisionAllegoryToDistributiveAllegory : DivisionAllegory >-> DistributiveAllegory.

6.4.1 The Division Allegory of Binary Relations

Our implementation of the left residual follows Lemma 3.6.1.

Definition LeftResidual_Rel (a b c : FNTDType): Rel a c → Rel b c → Rel a b := fun (S : Rel a c) (R : Rel b c) ⇒ Complement_Rel _ _ (Comp_Rel _ _ _ (Complement_Rel _ _ S) (Converse_Rel _ _ R)).

Instance MyRelDivisionAllegorySig : DivisionAllegorySig MyRelDistributiveAllegory := {
    leftResidual := LeftResidual_Rel
}.

In order to instantiate the class *DivisionAllegory* by set-theoretic relations we need the following lemma. It relates the two Boolean valued function $existsb$ and $forallb$ on list from the Coq library.

Lemma negb_existsb: forall (A : Type) (f : A → bool) (l : list A), negb (existsb f l) = forallb (fun x ⇒ negb (f x)) l.

Now we are ready to create the instance.

Instance MyRelDivisionAllegory : DivisionAllegory MyRelDivisionAllegorySig.

6.5 Implementation of Heyting Categories

Following Definition 3.5.1 the class *HeytingCategory* requires a division allegory and a greatest element for each hom-set.

Class HeytingCategory '(DA : DivisionAllegory)
(PCS : forall a b : Obj, PCSig (A := Mor a b))
(PCL : forall a b : Obj, PCLattice (L a b) (PCS a b))
(PCLE : forall a b : Obj, PCLELattice (PCL a b) (LEL a b)).

Below we have listed the Coq version of Lemma 3.5.2, Lemma 3.5.3 and an additional property that turned out to be useful in other proofs.

Lemma RtoLR '{HC : HeytingCategory} : forall a b: Obj (Q : Mor a b), Q ⊑ Q ∘ One.

Lemma extra_lemma '{HC : HeytingCategory} : forall a b: Obj (A : Mor a b), A ⊑ One ∘ A.

Lemma extra_lemma2 '{HC : HeytingCategory} : forall a b: Obj (A : Mor a b), A ˘ ⊑ A ˘ ∘ One.

Lemma extra_lemma3 '{HC : HeytingCategory} : forall a: Obj (A : Mor a a), A ∘ A ⊑ One ∘ A.

Lemma extra_lemma4 '{HC : HeytingCategory} : forall a: Obj (A : Mor a a), A ∘ A ⊑ A ∘ One.

Theorem P422_2v '{HC : HeytingCategory} : foralla b : Obj (Q R: Mor a b), univalent Q → R ⊑ Q → Q ∘ (One : Mor b b) ⊑ R ∘ (One : Mor b b) → R = Q.

Finally, we establish the coercion between Heyting categories and division allegories.

Instance HeytingCategoryToDivisionAllegory '(HC : HeytingCategory) : DivisionAllegory _.

Coercion HeytingCategoryToDivisionAllegory : HeytingCategory >-> DivisionAllegory.

6.5.1 The Heyting Category of Binary Relations

This time it is sufficient to provide the corresponding instance declaration. No proof is needed.

Instance MyRelHeytingCategory : HeytingCategory MyRelDivisionAllegory MyRelPCSig MyRelPCLattice MyRelPCLELattice.

6.6 Implementation of Schröder Categories

In order to implement Definition 3.6.1 we just need to provide a Boolean algebra structure on each hom-set.

Class SchroderCategory '(HC : HeytingCategory) (BA : forall a b : Obj, BooleanAlgebra (PCLE a b)).

Lemma ConvNeg '{S : SchroderCategory} a b : Obj : forall (R : Mor a b), R~˘ = R˘~.

The Tarski rule that we define chapter 3 (Definition 3.6.2) can be defined as follows.

Class TarskiRule '(SC : SchroderCategory) := {
    tarski_axiom : forall (a b c d : Obj) (R : Mor a b), R <> Zero →
      (One : Mor c a) ∘ R ∘ (One : Mor b d) = (One : Mor c d)
}.

Next, we establish the coercion between Schröder and Heyting categories.

Instance SchroderCategoryToHeytingCategory '(SC : SchroderCategory) : HeytingCategory _ _ _ _.

Coercion SchroderCategoryToHeytingCategory : SchroderCategory >-> HeytingCategory.

6.6.1 The Schröder Category of Binary Relations

Since we did not use any additional signature in the Coq implementation of a Schröder category we can provide the instance declaration immediately as follows:

Instance MyRelSchroderCategory : SchroderCategory MyRelHeytingCategory MyRelBooleanAlgebra.

Next we will prove that our implementation of binary relations also satisfies the Tarski rule. Similar to other instance declarations and lemmas we also omit the body.

Instance MyRelTarskiRule : TarskiRule MyRelSchroderCategory.

6.7 Implementation of a Unit

In order to implement a predicate indicating that a Heyting category has a unit object we will require the category and the unit object as parameters of the predicate. The following declaration and Definition 3.11.1 are analog. After that, we also provide an implementation of Lemma 3.11.1.

Definition hasUnit '(HC : HeytingCategory) (one : Obj) : Prop := (One : Mor one one) = id ∧ forall (a : Obj), total (One : Mor a one).

Lemma CompLL '{HC : HeytingCategory} {one : Obj} : hasUnit HC one → forall {a : Obj}, (One : Mor a one) ∘ (One : Mor one a) = One.

6.7.1 The Unit Object of Binary Relations

The abstract declaration of the predicate *hasUnit* uses an object as parameter. Therefore, we need to provide an instance of $FNTDType$ that serves as the unit. Coq already has a singleton data type called unit with element *tt*, i.e., $tt : unit$. Coq also provides decidability for the *unit* data type. It remains to show two properties. First, we need a lemma showing

that all elements of *unit* are contained in $[tt]$, and then we need to verify that $[tt]$ is not empty.

Lemma Finite_proof : forall(x : unit), In x [tt].

Lemma Empty_proof : [tt] <> [].

Now we can define a $FNTDType$ element based on *unit*.

Instance myOne : FNTDType := {
    A := unit;
    elements := [tt];
    finite_pr := Finite_proof;
    non_empty_pr := Empty_proof;
    Deq := unit_eqdec
}.

Finally, we can show that our concrete Heyting category has a unit.

Theorem MyRelhasUnit : hasUnit MyRelHeytingCategory myOne.

6.8 Implementation of Cardinality Functions

Before we can implement cardinality functions we need to define a class for monoids. We provide a signature for plus inside the class declaration. The appropriate axioms have been added as we need those to prove several lemmas. We use the infix operator $+$. We also define a coercion between the monoids and the underlying type $A1$ of the monoid. This allows us to treat the monoid as $A1$, i.e., a notation $x : M$ where M is a monoid instead of $x : A1$ becomes possible.

Class Monoid :={
    A1 : Type;
    zero : A1;
    plusM : A1 → A1 → A1;
    left_neutrality : forall x, plusM zero x = x;

right_neutrality : forall x, plusM x zero = x;
associativity : forall x y z, plusM x (plusM y z) = plusM (plusM x y) z;
commutativity : forall x y, plusM x y= plusM y x
}.

Coercion A1 : Monoid >-> Sortclass.

Infix "+" := (plusM) (at level 55, right associativity).

Now we need to declare a recursive function for multiplication as defined in Definition 3.8.2. Our declaration is given below.

Fixpoint nmult {M: Monoid} (n : nat) : A1 → A1 :=
match n with
| 0 ⇒ fun _ ⇒ zero
| S n ⇒ fun x ⇒ (nmult n x) x
end.

We use pattern matching on the natural number *n* in the function *nmult*. There are two cases for a natural number. Either the number is 0 or this number is the successor of the previous number. If the number is 0 then it will return *zero* otherwise function call itself recursively.

Now we are going to implement ordered monoids. In that class, we require monotonicity of *plus* as an axiom. The declaration is given below.

Class OrderedMonoid (M : Monoid) (OS : OrderSig (A := M)) (O : Order OS) :={
le_axiom_monoid : forall x, zero ⊑ x;
plus_mono : forall (x1 x2 y1 y2 : M), x1 ⊑ x2 → y1 ⊑ y2 → x1 + y1 ⊑ x2 + y2
}.

In the following we have given several lemmas related to ordered momoids.
Lemma MPlus '(OM : OrderedMonoid): forall (x y : M), x ⊑ x + y.

Lemma MPlus '(OM : OrderedMonoid): forall (x y : M), x ⊑ x + y.

Lemma twomult '(OM : OrderedMonoid) : forall (x :M), nmult 2 x = x + x.

Lemma nmmult '(OM : OrderedMonoid) : forall (n : nat) (x y:M), (nmult n x) + (nmult n y)= nmult n (x + y).

Lemma nmultMono '(OM : OrderedMonoid) : forall (n : nat) (x y:M), x ⊑ y → nmult n x ⊑ nmult n y.

Lemma singleMult '(OM : OrderedMonoid) : forall (k : nat) (x : M), x + nmult k x = nmult (k + 1) x.

In order to implement cardinality functions we need to provide a distributive allegory, an ordered monoid and a signature for the cardinality function. The following declaration is an exact implementation of Definition 3.7.1.

Definition hasCardinality '(DA : DistributiveAllegory)
'(OM : OrderedMonoid)
(Card : forall {a b : Obj}, Mor a b → A1) : Prop :=

(forall (x y : Obj) (R : Mor x y), (Card R) = zero <-> R = Zero)
∧ *(forall (x y : Obj) (R : Mor x y), (Card R) = (Card (R˘)))*
∧ *(forall (x y : Obj) (R S : Mor x y), (Card (R ⊔ S)) (Card (R ⊓ S))= (Card R) + (Card S))*
∧ *(forall (x y z : Obj) (Q : Mor z x) (R : Mor x y) (S : Mor z y), univalent Q → (Card (R ⊓ (Q˘ ∘ S)))⊑ (Card ((Q ∘ R) ⊓ S)))*
∧ *(forall (x y z : Obj) (Q : Mor z x) (R : Mor x y) (S : Mor z y), univalent Q → (Card (Q ⊓ (S ∘ R˘)))⊑ (Card ((Q ∘ R) ⊓ S))).*

We have defined a new tactic in order to unfold the definition of a cardinality function. It replaces the an assumption of the form H : *hasCardinality DA OM f* by the the individual axioms as separate assumptions named $C1, C2, C3, C_4a$ and C_4b.

Ltac destCardinality H := unfold hasCardinality in H; destruct H as [C1 H]; destruct H as [C2 H]; destruct H as [C3 H]; destruct H as [C_4a C_4b].

We proved Lemma 3.7.1 to 3.7.5 in Coq. To prove Lemma 3.7.2 we needed to prove fol-

lowing lemma.

Theorem hC42_Sub '{DA : DistributiveAllegory} '{OM : OrderedMonoid} {Card : forall (a b : Obj),Mor a b → A1} : hasCardinality DA OM Card → forall (x y z : Obj) (Q : Mor x y) (R : Mor y z) (S : Mor x z), (univalent Q) ∧ (univalent R) → Card _ _ (Q ∘ R ⊓ S) = Card _ _ (Q ⊓ S ∘ R˘).

6.8.1 The Cardinality of Binary Relations

In order to define a cardinality function for set-theoretic relations need to provide an instance of the class *Monoid*. In this case, the type will the type *nat* of natural numbers. Coq already provides a module called *Arith* where we get all the properties that we need for our instance declaration.

```
Instance myMonoid : Monoid := {
        A1 := nat;
        zero := 0;
        plusM := plus;
        left_neutrality := plus_O_n;
        right_neutrality := fun x : nat ⇒ (eq_sym) (plus_n_O x);
        associativity := plus_assoc;
        commutativity := plus_comm ;
}.
```

Next, we need to provide an instance of the signature of an order, and in our case, this will be the order of natural number. Then we need to provide an instance of class *Order*. these two instance to make the monoid of natural numbers an instance of the *OrderedMonoid* class. The implementation of this three instance as follows.

```
Instance myMonoidOrderSig : (OrderSig (A := myMonoid)) := {
        leq := le
}.

Instance myMonoidOrder : Order myMonoidOrderSig := {
        leq_refl := le_refl;
```

```
    leq_trans := le_trans;
    leq_anti := le_antisym;
}.

Instance myOrderedMonoid : (OrderedMonoid myMonoid myMonoidOrderSig myMonoidOrder)
:= {
    le_axiom_monoid := le_0_n;
    plus_mono := plus_le_compat;
}.
```

Now we need to implement the cardinality function for set-theoretic relations. Our definition will return total number pairs in a relation.

```
Definition inner {a : Type}: (a → bool) → nat → a → nat := fun p n x ⇒ if p x then n+1
else n.

Definition myCard (a b : FNTDType) : Rel a b → myMonoid := fun R ⇒ fold_left (inner
(prod_curry R)) (nodup (pairDeq Deq Deq) (list_prod elements elements)) 0.
```

Several lemmas and definitions are required to prove that cardinality function satisfies the required axioms.

```
Lemma fold_left_plus {a : Type}: forall (l : list a) (f : nat → a → nat) (m : nat), (forall (x :
a) (n : nat), f (m + n) x = m + f n x) → forall (n : nat), fold_left f l (m+n) = m + (fold_left f
l n).

Lemma inner_prop {a : Type}: forall (m : nat) (p : a → bool) (x : a) (n : nat), inner p (m +
n) x = m + inner p n x.

Lemma fold_inner_plus {a : Type}: forall (l : list a) (m n : nat) (p : a → bool), fold_left
(inner p) l (m+n) = m + fold_left (inner p) l n.

Lemma empty_has_none {a : Type}: forall (l : list a) (p : a → bool), fold_left (inner p) l 0
= 0 → (forall (x : a), In x l → p x = false).

Lemma has_none_empty {a : Type}: forall (l : list a) (p : a → bool), (forall x, In x l → p x
```

= false) → fold_left (inner p) l 0 = 0.

Lemma in_prop_preserved {a b : Type} : forall (f : a → b) (l : list a) (l1 l2 : list b) (y : a) (def : a → Prop), (forall (x y : a), def x → def y → f x = f y → x = y) → (forall (x : a), def x → In x (y :: l) → In (f x) (l1 ++ f y :: l2)) → NoDup (y::l) → def y → forall (x : a), def x → In x l → In (f x) (l1 ++ l2).

Lemma myCard_Ext {a b : Type}: forall (f : a → b) (l1 : list a) (p : a → bool) (q : b → bool),(forall (x y : a), p x = true → p y = true → f x = f y → x = y) → (forall (x : a), p x = true → q (f x) = true) → NoDup l1 → forall (l2 : list b), (forall (x : a), p x = true → In x l1 → In (f x) l2) → fold_left (inner p) l1 0 ⇐ fold_left (inner q) l2 0.

*Definition swap {a b : Type}: a*b → b*a := fun p ⇒ (snd p,fst p).*

*Lemma swap_inj {a b : Type}: forall (p1 p2 : a*b), swap p1 = swap p2 → p1 = p2.*

*Lemma swap_list {a b : Type}: forall (p : a*b) (l1 : list a) (l2 : list b), In p (list_prod l1 l2) → In (swap p) (list_prod l2 l1).*

*Definition fProp4 a b c : Type (l : list a) (Q : Rel a b) (S : Rel a c) (default : a): b*c → a*c := fun p ⇒ match (find(fun (x : a) ⇒ (Q x (fst p)) && (S x (snd p))) l) with*
| Some a1 ⇒ (a1, snd p)
| None ⇒ (default, snd p)
end.

Lemma uni_concrete {a b : FNTDType}: forall (Q : Rel a b) (x : a) (y0 y1 : b), univalent (A:=MyRelAllegory) Q Q x y0 = true Q x y1 = true y0 = y1.

*Lemma fProp4_Prop1 {a b c : FNTDType}: forall (Q : Rel a b) (R : Rel b c) (S : Rel a c) (default : a), univalent (A:=MyRelAllegory) Q → forall p : b*c, prod_curry (Meet_Rel R (Comp_Rel _ _ _ (Converse_Rel _ _ Q) S)) p = true → prod_curry (Meet_Rel (Comp_Rel _ _ _ Q R) S) (fProp4 elements Q S default p) = true.*

*Lemma fProp4_rel_inj {a b c : FNTDType}: forall (Q : Rel a b) (R : Rel b c) (S : Rel a c) (default : a), univalent (A:=MyRelAllegory) Q → forall (p1 p2 : b*c),(prod_curry (Meet_Rel R (Comp_Rel _ _ _ (Converse_Rel _ _ Q) S))) p1 = true → (prod_curry (Meet_Rel*

R (Comp_Rel _ _ _ (Converse_Rel _ _ Q) S))) p2 = true → fProp4 elements Q S default p1 = fProp4 elements Q S default p2 → p1 = p2.

*Definition fProp5 {a b c : FNTDType} (l : list c) (R : Rel b c) (S : Rel a c) (default : c): a*b → a*c := fun (p : a*b) ⇒ swap (fProp4 l (Converse_Rel _ _ R) (Converse_Rel _ _ S) default (swap p)).*

*Lemma fProp5_Prop1 {a b c : FNTDType}: forall (Q : Rel a b) (R : Rel b c) (S : Rel a c) (default : c), univalent (A:=MyRelAllegory) Q → forall p : a*b, prod_curry (Meet_Rel Q (Comp_Rel _ _ _ S (Converse_Rel _ _ R))) p = true → prod_curry (Meet_Rel (Comp_Rel _ _ _ Q R) S) (swap (fProp4 elements (Converse_Rel _ _ R) (Converse_Rel _ _ S) default (swap p))) = true.*

*Lemma fProp5_rel_inj {a b c : FNTDType}: forall (Q : Rel a b) (R : Rel b c) (S : Rel a c) (default : c), univalent (A:=MyRelAllegory) Q → forall (p1 p2 : a*b), (prod_curry (Meet_Rel Q (Comp_Rel _ _ _ S (Converse_Rel _ _ R)))) p1 = true → (prod_curry (Meet_Rel Q (Comp_Rel _ _ _ S (Converse_Rel _ _ R)))) p2 = true → (swap (fProp4 elements (Converse_Rel _ _ R) (Converse_Rel _ _ S) default (swap p1))) = (swap (fProp4 elements (Converse_Rel _ _ R) (Converse_Rel _ _ S) default (swap p2)))→ p1 = p2.*

Lemma fExt {a : Type}: forall (f g h : nat → a → nat) (l : list a), (forall (x : a), f 0 x = g 0 x + h 0 x) → (forall (x : a) (n m : nat), f (m + n) x = m + f n x) → (forall (x : a) (n m : nat), g (m + n) x = m + g n x) → (forall (x : a) (n m : nat), h (m + n) x = m + h n x) → fold_left f l 0 = fold_left g l 0 + fold_left h l 0.

Finally, we declare an instance of *hasCardinality* called *MyRelhasCardinality*. To prove the required properties, we use all these lemmas that we have stated previously.

Theorem MyRelhasCardinality : hasCardinality MyRelDistributiveAllegory myOrderedMonoid myCard.

6.9 Implementation of Atom and Edge

Using the properties of Heyting categories we define the predicate *isAtom* as follows:

Definition isAtom '{HC : HeytingCategory} {a b: Obj} (A : Mor a b) := (A <> Zero) ∧ ((A

$^\smile \circ One \circ A) \sqsubseteq id) \wedge ((A \circ One \circ A^\smile) \sqsubseteq id)$.

Within our algorithms we need to extract an atom from a relation using a function called *atom*. The following predicate tells us whether this is possible and what properties are satisfied by *atom R*.

Definition hasAtom '(HC : HeytingCategory) (atom : forall {a b : Obj}, Mor a b → Mor a b) : Prop := forall (a b: Obj) (R : Mor a b), R <> Zero → ((isAtom (atom R)) ∧ ((atom R) ⊑ R)).

Now we define the edge predicate according to Definition 3.10.2.

Definition edge '{HC : HeytingCategory} {atom : forall {a b : Obj}, Mor a b → Mor a b} (hA : hasAtom HC (@atom)) {a : Obj} : Mor a a → Mor a a := fun (R : Mor a a) ⇒ (atom R) ⊓ (atom R)^\smile .

As before we define a tactic to destruct the predicate *isAtom*.

Ltac destAtom H := unfold isAtom in H; destruct H as [atomAxiom1 H]; destruct H as [atomAxiom2 atomAxiom3].

Now we provide some lemmas about atoms.

Lemma iA3_univalence '{HC : HeytingCategory} : forall {a b: Obj} (A : Mor a b), isAtom A → (A^\smile ∘ A) ⊑ id.

Lemma iA3_injectivity '{HC : HeytingCategory} : forall {a b: Obj} (A : Mor a b), isAtom A → (A ∘ A^\smile) ⊑ id.

Lemma iA3_transitivity '{HC : HeytingCategory} : forall {a: Obj} (A : Mor a a), isAtom A → (A ∘ A) ⊑ A.

Lemma iA3_2_4 '{HC : HeytingCategory} : forall {a: Obj} (A : Mor a a), isAtom A → (A ∘ A) ⊑ id.

Similarly, we prove some lemmas about edges.

Lemma iA33_symmetry '{HC : HeytingCategory} {atom : forall {a b : Obj}, Mor a b → Mor a b} {hA : hasAtom HC (@atom)} {a : Obj} : forall (R : Mor a a) ,(edge hA R) = (edge hA R)˘.

Lemma iA33_2 '{HC : HeytingCategory} {atom : forall {a b : Obj}, Mor a b → Mor a b} {hA : hasAtom HC (@atom)} {a : Obj} : forall (R : Mor a a), isAtom (atom R) → (edge hA R) ∘ (edge hA R) ⊑ id.

Lemma atomIsMap '(TR : TarskiRule) (atom : forall {a b : Obj}, Mor a b → Mor a b) (hA : hasAtom HC (@atom)) {a b one: Obj} : forall (R : Mor a b), hasUnit HC one wedge R <> Zero → map_Rel (((atom R) ∘ (One : Mor b one)) ˘).

Lemma atomIsMap1 '(TR : TarskiRule) (atom : forall {a b: Obj}, Mor a b → Mor a b) (hA : hasAtom HC (@atom)) {a one: Obj} : forall (R : Mor a a), hasUnit HC one wedge R <> Zero → map_Rel (((atom R)˘ ∘ (One : Mor a one)) ˘).

6.9.1 Implementation of Atoms for Binary Relations

Our implementation of the function *atom* for set-theoretic relations simply returns the first pair. By the first pair we mean the first pair that is in the relation by using the two lists of elements provided by each finite type.

Definition myRelAtom : forall (a b : FNTDType), Rel a b → Rel a b := fun a b R ⇒
*match (find (fun (p : (a*b)) ⇒ ((prod_curry R) p)) (list_prod elements elements)) with*
| Some p ⇒ fun x y ⇒ CDeq (fst p) x &&& CDeq (snd p) y
| None ⇒ R
end.

Finally, we can define an instance of the class *hasAtom* where we prove that our implementation of *atom* satisfies all required axioms.

Theorem MyRelhasAtom : hasAtom MyRelHeytingCategory myRelAtom.

6.10 Implementation of Direct Products

Similar to the previous section, we implement products as a predicate on Heyting algebras. This predicate takes as input the category in question and three functions. The first function maps two objects to the direct product, and the second and third functions return the first and second projection from the direct product to the objects in question, respectively. In other words the predicate *hasProduct* indicates that every pair of objects has a direct product. Please note that the implementation of *hasProduct* is an immediately implementation of Definition 3.8.1.

Definition hasProduct '(HC : HeytingCategory)
(ProdObj : Obj → Obj → Obj)
(π : forall (a b : Obj), Mor (ProdObj a b) a)
(ρ : forall (a b : Obj), Mor (ProdObj a b) b) : Prop :=

(forall (a b : Obj), (π a b)˘ ∘ (π a b) = id)
∧ *(forall (a b : Obj), (ρ a b)˘ ∘ (ρ a b) = id)*
∧ *(forall (a b : Obj), ((π a b) ∘ (π a b)˘) ⊓ ((ρ a b) ∘ (ρ a b)˘) = id)*
∧ *(forall (a b : Obj), (π a b) ˘ ∘ (ρ a b) = One).*

We need a lemma which states that the composition of the converse of ρ and π is also equal to the greatest element.

Theorem ConverseAxiom4 '{HC : HeytingCategory} : forall (ProdObj : Obj → Obj → Obj) (π : forall (a b : Obj), Mor (ProdObj a b) a) (ρ : forall (a b : Obj), Mor (ProdObj a b) b), hasProduct HC ProdObj π ρ → forall(a b : Obj), (ρ a b)˘ ∘ (π a b) = One.

In the Coq implementation we have also provided proofs of the Lemmas 3.8.1, 3.8.2 and 3.8.3 which We omit them here. We did not define the strict fork operation, strict join operation, and Kronecker product. Therefore, we will use their definition the corresponding symbol.

6.10.1 The Direct Product for Binary Relations

First, we need to define the product object. Since objects in our Heyting category of finite relations are instances of the class $FNTDType$ we have to provide an appropriate type

with a decidable equality, a list of its elements, and proofs that this list contains all elements and is not empty. The type will be the type $a \times b$ of pairs from a and b, of course. In order to produce a list of its elements we apply the Coq function *list_prod* to the two lists of elements in a and b. Below we have listed the two lemmas required to show that *list_prod elements elements* satisfies the required properties. Furthermore, we provide the declaration of the function *pairDeq* that maps proofs of the detectability of the equality on a and b to a proof of the decidability of the equality on $a \times b$. For details of this function we refer to the implementation.

```
Lemma Finite_proof_myProd {a b : FNTDType} : forall(x : a * b), In x (list_prod elements elements).

Lemma empty_prod {a b : Type} (l1 : list a) (l2 : list b) : l1 <> [] → l2 <> [] → (list_prod l1 l2) <> [].

Definition pairDeq {A B : Type}: (forall x y : A, {x = y} + {x <> y}) → (forall x y : B, {x = y} + {x <> y}) → forall x y : A * B, {x = y} + {x <> y}.
```

Now we are ready to make the type $a \times b$ an instance of $FNTDType$.

```
Instance myProdObj (a b : FNTDType) : FNTDType := {
    A := a * b;
    elements := list_prod elements elements;
    finite_pr := Finite_proof_myProd;
    non_empty_pr := empty_prod elements elements non_empty_pr non_empty_pr;
    Deq := pairDeq Deq Deq
}.
```

Below we have listed the definition of the first and second projection as a set theoretic relation.

```
Definition myPi (a b : FNTDType) : Rel (myProdObj a b) a := fun p z ⇒ CDeq (fst p) z.

Definition myRho (a b : FNTDType) : Rel (myProdObj a b) b := fun p z ⇒ CDeq (snd p) z.
```

The following theorem shows that the Heyting category of set theoretic relations with the

definitions above has direct products.

Theorem MyRelhasProduct : hasProduct MyRelHeytingCategory myProdObj myPi myRho.

6.11 Implementation of Direct Sum

Similar to the previous section we define a predicate *hasSum* that indicates that a Heyting category has a direct sum for each pair of objects. In its implementation we follow Definition 3.9.1.

Definition hasSum '(HC : HeytingCategory)
(SumObj : Obj → Obj → Obj)
(ι : forall (a b : Obj), Mor a (SumObj a b))
(κ : forall (a b : Obj), Mor b (SumObj a b)) : Prop :=

(forall (a b : Obj), (ι a b) ∘ (ι a b)˘ = id)
∧ *(forall (a b : Obj), (κ a b) ∘ (κ a b)˘ = id)*
∧ *(forall (a b : Obj), ((ι a b)˘ ∘ (ι a b)) ⊔ ((κ a b)˘ ∘ (κ a b)) = id)*
∧ *(forall (a b : Obj), (ι a b) ∘ (κ a b)˘ = Zero).*

Similar to projections we show that κ and converse of ι is equal to least element.

Theorem ConverseAxiom4_Sum '{HC : HeytingCategory} : forall (SumObj : Obj → Obj → Obj) (ι : forall (a b : Obj), Mor a (SumObj a b)) (κ : forall (a b : Obj), Mor b (SumObj a b)), hasSum HC SumObj ι κ → forall(a b : Obj), (κ a b) ∘ (ι a b)˘ = Zero.

We also proved Lemmas 3.9.1, 3.9.2 and 3.9.3 which we omit in this thesis.

6.11.1 The Direct Sum for Binary Relations

Similar to the direct product we have to create an instance of the class $FNTDType$ based on the sum $a + b$ of two types a and b. Unfortunately, Coq does not provide a function similar to *list_prod* for sums.

Definition SumProd {a b : Type} : list a → list b → list (a + b) := fun xs ys ⇒ (map inl xs)

++ (map inr ys).

Function *SumProd* takes two lists as a parameters and returns a list where the type of each element is the sum of provided types. Please note that *inl* and *inr* are the Coq implementations of the injections.

As before the following declarations are needed in order to make $a + b$ an instance of $FNTDType$, which follows immediately after.

Lemma in_sum {a b : Type} (l1:list a) (l2:list b) : (forall(x : a), In x l1) → (forall (y : b), In y l2) → forall (z : a + b), In z (SumProd l1 l2).

Lemma empty_sum {a b : Type} (l1 : list a) (l2 : list b) : l1 <> [] → l2 <> [] → (SumProd l1 l2) <> [].

Definition sumDeq {A B : Type}: (forall x y : A, {x = y} + {x <> y}) → (forall x y : B, {x = y} + {x <> y})→ forall x y : A + B, {x = y} + {x <> y}.

```
Instance mySumObj (a b : FNTDType) : FNTDType := {
    A := a + b;
    elements := SumProd elements elements;
    finite_pr := in_sum elements elements finite_pr finite_pr;
    non_empty_pr := empty_sum elements elements non_empty_pr non_empty_pr;
    Deq := sumDeq Deq Deq
}.
```

After defining the injections as relations below we verify that the Heyting category of set theoretic relations has directed sums.

Definition myIota (a b : FNTDType) : Rel a (mySumObj a b) := fun z s ⇒ CDeq s (inl z).

Definition myKappa (a b : FNTDType) : Rel b (mySumObj a b) := fun z s ⇒ CDeq s (inr z).

Theorem MyRelhasSum : hasSum MyRelHeytingCategory mySumObj myIota myKappa.

6.12 Well-Founded Inclusion Order of Relations

In order to apply our algorithms to set theoretic relations, we have to show that the inclusion order and its reversed order are well-founded. The following two definitions provide the proof term for these facts.

Definition well_founded_Relation '(A : Allegory): Prop := forall (x y : Obj),well_founded (fun (R S : Mor x y) ⇒ R ⊏ S).

Definition well_founded_Relation_gr '(A : Allegory): Prop := forall (x y : Obj),well_founded (fun (R S : Mor x y) ⇒ R ⊐ S).

In order to implement the second definition we need several additional lemmas and theorems establishing the fact the converse of every well-founded relation on A is also well-founded if there is an order reversion and involutive function f on A. We refer to the Coq implementation for details.

6.12.1 Well-Founded Inclusion Order of Binary Relations

We would like to use the fact that the natural numbers are well-ordered while verifying that the inclusion order on set theoretic relations is also well-ordered. This will be possible since the cardinality function relates the inclusion order with the order on the natural numbers. For that purpose we have shown three lemmas relating a well-founded relation on the image of a function f to a well-founded relation on the domain of f.

Lemma Acc_invImage_f {A B : Type} {R : relation A} {S : relation B} {f : A → B} : (forall x y, R x y → S (f x) (f y)) → forall y, Acc S y → forall x:A, y = f x → Acc R x.

Lemma Acc_invImage {A B : Type} {R : relation A} {S : relation B} f : A → B : (forall x y, R x y → S (f x) (f y)) → forall x, Acc S (f x) → Acc R x.

Lemma wf_invImage {A B : Type} {R : relation A} {S : relation B} {f : A → B} : (forall x y, R x y → S (f x) (f y)) → well_founded S → well_founded R.

In order to use Lemma *wf_invImage* for the cardinality function we need to verify that if R is strictly included in S, then the cardinality of R is strictly smaller than the cardinality of S. Essential for this proof is to provide a pair p that is in S but not in R. All of this is done

in the following sequence of lemmas.

Lemma neq_Rel_find {x y : FNTDType} {R S : Rel x y} : R <> S → find (fun p ⇒ negb (eqb ((prod_curry R) p) ((prod_curry S) p))) (nodup (pairDeq Deq Deq) (list_prod elements elements)) <> None.

Lemma lt_Rel_Witness {x y : FNTDType} {R S : Rel x y} : R ⊏ S → exists a b, R a b = false ∧ S a b = true.

Lemma myCardSubMono {x y : FNTDType} {R S : Rel x y} : forall l, R ⊑ S → fold_left (inner (prod_curry R)) l 0 <= fold_left (inner (prod_curry S)) l 0.

Lemma ltTOle : (forall n m : nat, n < m <-> n <= m ∧ n <> m).

Lemma myCardSubStrictMono {x y : FNTDType} {R S : Rel x y} : forall l, R ⊑ S → (exists a b, In (a,b) l ∧ R a b = false ∧ S a b = true) → fold_left (inner (prod_curry R)) l 0 < fold_left (inner (prod_curry S)) l 0.

Lemma myCardStrictMono {x y : FNTDType} : forall (R S : Rel x y), R ⊏ S → myCard _ _ R < myCard _ _ S.

When we use *myCardStrictMono* as an argument for *wf_invImage*, then we will get a goal where we need to prove that $<$ is an well-order on the natural numbers. Coq provides Lemma *well_founded_ltof* which shows exactly that. Together this gives us the following:

Theorem MyRel_well_founded_Relation : well_founded_Relation MyRelAllegory.

Now we can use Lemma *wf_le_ge* to show the following:

Theorem MyRel_well_founded_Relation_GR : well_founded_Relation_gr MyRelAllegory.

6.13 Decidability of Equality of Relations

The algorithms will require that the equality on relations is decidable. Therefore, we would like to establish this property for our category of set theoretic relations. First, we define a

class that adds a proof of decidability to a Schröder category.

Class EqDec_eq_Rel '(SC : SchroderCategory) := eq_dec_Rel : forall {a b: Obj} (R S : Mor a b), {R = S} + {R <> S}.

The next class adds the decidability of the equality of the cardinality of two relations to a distributive allegory.

Class EqDecCard '(DA : DistributiveAllegory) '(OM : OrderedMonoid) (Card : forall (a b : Obj),Mor a b → A1) := Dec_Type : forall {a b: Obj} (R S : Mor a b), {Card _ _ R = Card _ _ S} + {Card _ _ R <> Card _ _ S}.

Similarly, the next class requires that it is decidable whether the cardinality of one relation is strictly smaller that the cardinality of another relation.

Class LtDec_eq_card '(DA : DistributiveAllegory) '(OM : OrderedMonoid) (Card : forall (a b : Obj),Mor a b → A1) := lt_dec_card : forall {a b: Obj} (R S : Mor a b), {Card _ _ R⊏ Card _ _ S} + {~(Card _ _ R ⊏ Card _ _ S)}.

Last but not least, the following class requires that less or equal than on the cardinality of two relations is decidable.

Class LECard '(DA : DistributiveAllegory) '(OM : OrderedMonoid) (Card : forall (a b : Obj),Mor a b → A1) := LE_Card_Axiom: forall {a b: Obj} (R S : Mor a b), (Card _ _ R ⊑ Card _ _ S) ∨ (Card _ _ S ⊑ Card _ _ R).

In order to show that the equality for finite set theoretic relations is decidable, we implement a function *findbool* that implements equality as a Boolean valued function.

Definition findbool {a b: FNTDType} : Rel a b → Rel a b → bool := fun R S ⇒ forallb (fun p : (a ∗ b) ⇒ eqb ((prod_curry R) p) ((prod_curry S) p)) (list_prod elements elements).

Using the previous function we can immediately show the next theorem and instantiate the class *EqDec_eq_Rel* from above.

Theorem RelationEqual {a b: FNTDType} (R : Rel a b) (S : Rel a b) : findbool R S = true

<-> R = S.

Instance MyRelEqDec_eq_Rel : EqDec_eq_Rel MyRelSchroderCategory.

Coq provides two theorems that show the decidability of $=$ and $<$ on natural numbers called *eq_nat_decide* and *lt_dec*. We use these theorem in the following instance declarations.

Instance MyRelEqDecCard : EqDecCard MyRelDistributiveAllegory myOrderedMonoid myCard.

Instance MyRelLtDec_eq_card : LtDec_eq_card MyRelDistributiveAllegory myOrdered-Monoid myCard.

In order to provide and instance of the class *LECard* we use almost same technique as in the instance declaration *MyRelLtDec_eq_card*. It requires just an additional case distinction.

Instance MyRelLECard : LECard MyRelDistributiveAllegory myOrderedMonoid myCard.

Chapter 7

Implementation of Approximation Algorithms

Using the framework from the previous chapter we are going to implement each algorithm in Coq, verify its correctness, and apply it to an example in this chapter.

7.1 Vertex Covers Problem

7.1.1 Abstract Implementation of the Algorithm

In this section, we implemented the recursive version of the algorithm outlined in the in Section 4.1.3 in Coq. Since all algorithms in Coq have to terminate we first have to establish this property by requiring an adequate category of relations.

We have decided to list those requirements by defining variables or parameters of the Coq module that provide the corresponding property. In the following we have listed the essential requirements. For a full list we refer to the Coq implementation.

Variable SC : SchroderCategory HC BA.
Variable TR : TarskiRule SC.
Variable hU : hasUnit HC one.
Variable Card : forall {a b : Obj}, Mor a b → A1.
Variable hC : hasCardinality DA OM Card.
Variable hA : hasAtom HC (@atom).
Variable eqDec : EqDec_eq_Rel SC.
Variable WFR : well_founded_Relation A.

To summarize the list above we require a Schröder that satisfies the Tarski rule, has a unit, has a cardinality and atom function, and for which equality is decidable and the order is well-founded.

The next two lemmas show that each recursive call will use a strictly smaller argument and that the property of S being symmetric is preserved.

Lemma Decreasing : forall {a : Obj} (S e: Mor a a), S ˘ = S → S <> Zero → e = edge hA S → S ⊓ ((e ∘ One) ⊔ (One ∘ e)) ~⊏ S.

Lemma SymPreserved : forall {a : Obj} (S e: Mor a a), S˘ = S → e = edge hA S → (S ⊓ (e ∘ One ⊔ One ∘ e)~) ˘ = S ⊓ (e ∘ One ⊔ One ∘ e)~.

The following code implements the algorithm in Coq. Please note that we use the two lemmas above as parameters of *Fix* in order to guarantee termination.

```
Definition vertexCover {a : Obj} (S : Mor a a) : S˘ = S → Mor a one :=
    Fix (WFR a a) (fun (S : Mor a a) ⇒ S˘ = S → Mor a one)
    (fun (S : Mor a a)
     (vertexCover : forall R : Mor a a, R ⊏ S → R˘ = R → Mor a one)
     ⇒ match eqDec _ _ S Zero with
       | left _ ⇒ fun _ ⇒ Zero
       | right n ⇒ fun sym ⇒ let e := edge hA S in
           let S' := S ⊓ (e ∘ One ⊔ One ∘ e)~in
            (e ∘ One) ⊔ vertexCover S' (Decreasing S e sym n (eq_refl e))
            (SymPreserved S e sym (eq_refl e))
    end ) S.
```

Please note that if all information and parameters concerning termination is removed from the code we obtain exactly the recursive algorithm from Section 4.1.3.

After proving some auxiliary lemmas we prove the following theorem which is the Coq analog of Theorem 4.1.5, i.e., proving the correctness of the algorithm.

Theorem approxVC {a : Obj} (R : Mor a a): forall (c : Mor a one) (p : R˘ = R), c =

vertexCover R p $\rightarrow$ *R* $\sqsubseteq$ *c* $\circ$ *One* $\sqcup$ *(c* $\circ$ *One)*$^\smile$ $\wedge$ *forall (d : Mor a one), R* $\sqsubseteq$ *d* $\circ$ *One* $\sqcup$ *(d* $\circ$ *One)*$^\smile$ $\rightarrow$ *Card _ _ c* $\sqsubseteq$ *nmult 2 (Card _ _ d).*

In order to prove this theorem, we use *well_founded_induction* as discussed in Chapter 5.

7.1.2 Example

In this section, we implement Example 4.1.2. First, we define a simple enumeration type *Nodes* in Coq with elements $a, \ldots, h$. This data type will serve as the set of nodes of the graph. Then we create an instance of the class $FNTDType$ based on *Nodes*. Last but not least, we define the graph as a relation G. For details of those definitions we refer to the Coq implementation. The following Figure 7.1 represent graph G.

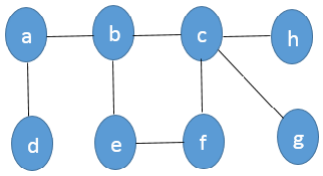


Figure 7.1: Example of Graph

We have to verify that G is indeed symmetric in order to satisfy the pre-condition of the vertex cover algorithm.

Lemma GraphIssymmetric : Converse_Rel _ _ G = G.

In the next step we have to refine our abstract algorithm to the concrete category of finite set theoretic relations by instantiating each parameter of the module appropriately. Then we can call the algorithm. Figure 7.2 shows the instantiation of the parameters and also the output for our example.

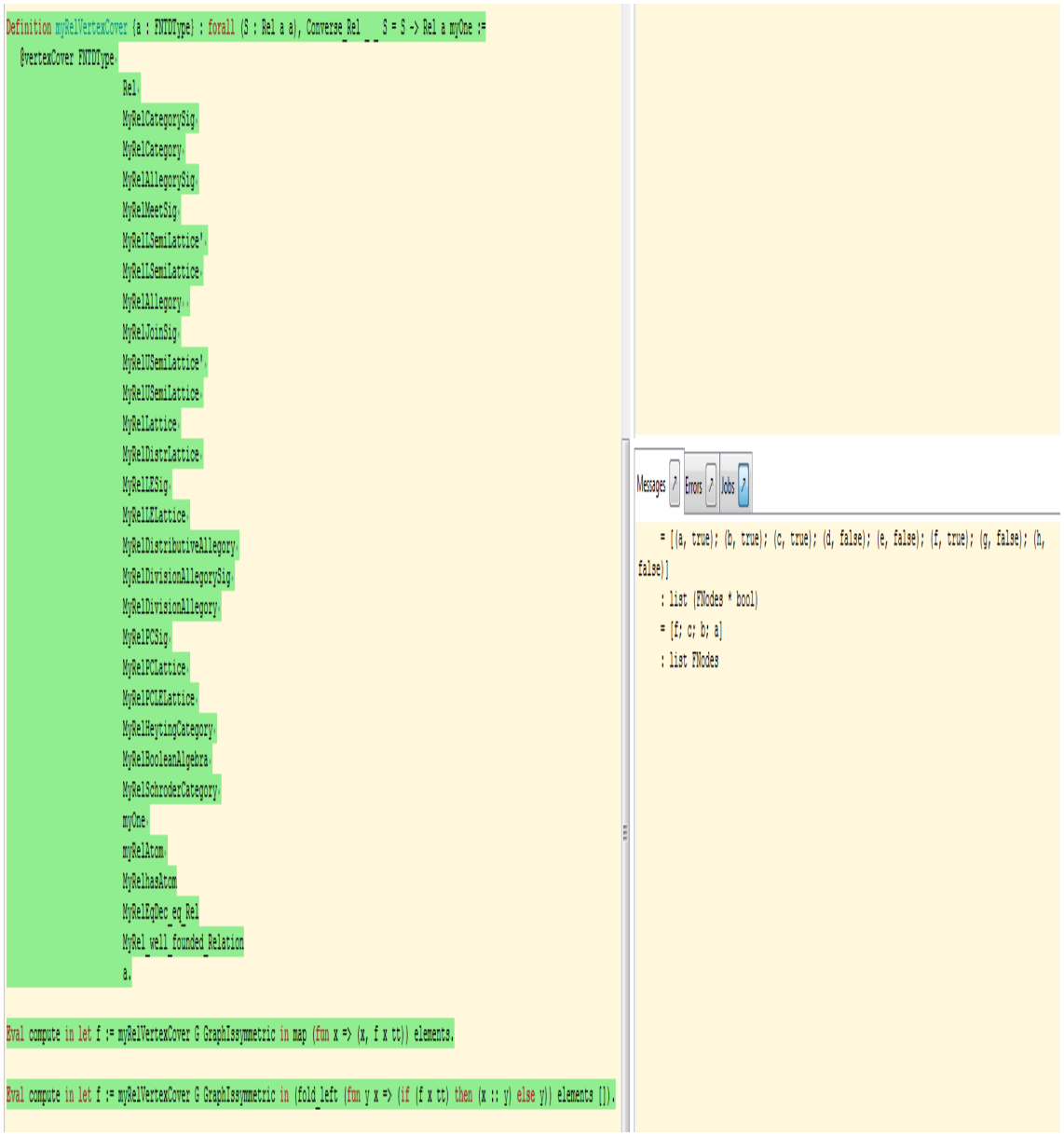


Figure 7.2: Implementation and Output of Vertex Cover

The first call returns the vertex cover as list representation of the vector. The second call simply returns the vertex cover as a list.

7.2 Hitting Sets

7.2.1 Abstract Implementation of the Algorithm

The relational algorithm that we use for the hitting sets problem takes exactly the same parameters as the algorithm that we use for vertex cover. Similar to the vertex cover problem, we need to prove that our program will terminate. Therefore we proved Lemma 4.2.1, and we do not need to show any additional properties as for the previous algorithm since the precondition is not related to any relation of the recursive call. First we show the termination lemma and then we define the algorithm.

Lemma Decreasing_HS : forall {a b: Obj} (s P: Mor b one) (I: Mor a b), id ⊑ I˘ ∘ I → s <> Zero → P = @atom _ _ s → s ⊓ (I˘ ∘ I ∘ P) ~⊏ s.

Definition hittingSets' {a b: Obj} (I : Mor a b) (surj : id ⊑ I ˘ ∘ I) (c : Mor a one) (s : Mor b one) : Mor a one :=

```
Fix (WFR b one) (fun _ ⇒ Mor a one → Mor a one)
(fun (s : Mor b one)
 (hittingSets : forall (s' : Mor b one), s' ⊏ s → Mor a one → Mor a one)
 ⇒ match eqDec _ _ s Zero with
   | left _ ⇒ fun c ⇒ c
   | right n ⇒ fun c ⇒
     let p := @atom b one s in
     let s' := s ⊓ (I˘ ∘ I ∘ p)~in
       hittingSets s' (Decreasing_HS s p I surj n (eq_refl p)) (c ⊓ I ∘ p)
end) s c.
```

Definition hittingSets {a b: Obj} (I : Mor a b) (surj : id ⊑ I ˘ ∘ I) : Mor a one := hittingSets' I surj Zero One.

Similar to the previous algorithm, we will get exactly the algorithm presented in Section 4.2 if we remove all information and parameters regarding termination.

In order to prove the correctness of the program, we needed to prove Theorem 4.2.5 and 4.2.6 in Coq. We used one auxiliary lemma to prove those theorems. The declaration of that lemma and the theorems in Coq is given below:

Lemma hittingSets'_eq {a b: Obj} (I : Mor a b) (surj : id ⊑ I ˘ ∘ I) : forall (c : Mor a one) (s : Mor b one), hittingSets' I surj c s = if eqDec b one s Zero then c else hittingSets' I surj (c ⊔ I ∘ @atom b one s) (s ⊓ (I ˘ ∘ I ∘ @atom b one s)~).

Theorem approxHS_PartA {a b: Obj} (I : Mor a b) (surj : id ⊑ I ˘ ∘ I) (k :nat) (preK : forall (p : Mor b one), injective p → Card a one (I ∘ p) ⊑ nmult k (Card one one id)): forall (s : Mor b one) (c : Mor a one), s~⊑ I ˘ ∘ c → (forall (d : Mor a one), (I ˘ ∘ I ∘ s)~⊑ I ˘ ∘ d → Card _ _ c ⊑ nmult k (Card _ _ d)) → let c' := hittingSets' I surj c s in One = I ˘ ∘ c' ∧ forall (d : Mor a one), One ⊑ I ˘ ∘ d → Card _ _ c' ⊑ nmult k (Card _ _ d).

Theorem approxHS {a b: Obj} (I : Mor a b) (surj : id ⊑ I ˘ ∘ I) (k :nat) (preK : forall (p : Mor b one), injective p → Card a one (I ∘ p) ⊑ nmult k (Card one one id)) : let c := hittingSets I surj in One = I ˘ ∘ c ∧ forall (d : Mor a one), One ⊑ I ˘ ∘ d → Card _ _ c ⊑ nmult k (Card _ _ d).

7.2.2 Example

In this section, we apply our approximation algorithm to a concrete hypergraph. The relation I is of type $X \to E$ for a graph $G = (X, E)$. So we need to define a data type for both nodes and edges. In our example we have a set of $Nodes$ with elements $N0, \ldots, N3$ and a set of $Edge$ with elements $E0, \ldots, E2$. After that we needed a create a instance of $FNTDType$ based on $Nodes$ and $Edge$. Similar to the previous example we define the graph by its incident relation that we have called G. The following Figure 7.3 is the graph representation of our graph G.

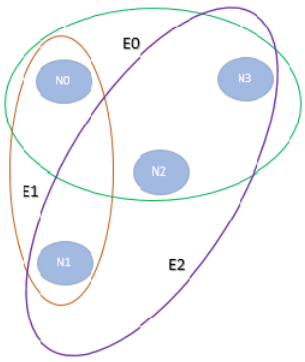


Figure 7.3: Example of Hyper Graph

In order to satisfy the pre-condition for the hitting set algorithm we need to prove our G is injective.

Lemma GraphIsInjective : ID_Rel _ ⊑ Comp_Rel _ _ _ (Converse_Rel _ _ G) G.

The following Figure 7.4 shows the instantiation of parameters and output of our example.

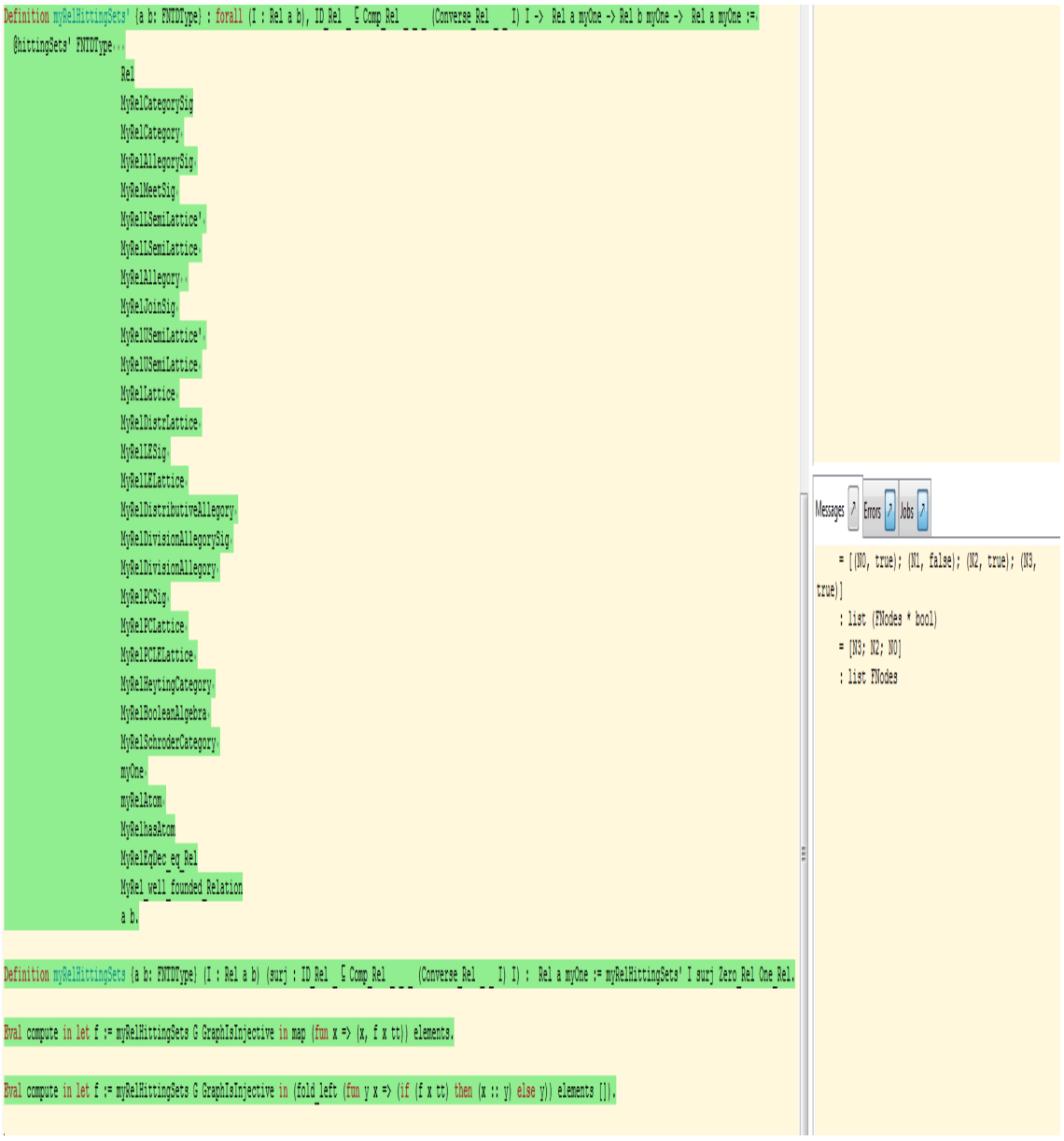


Figure 7.4: Declaration and Output of Hitting Sets

7.3 Maximum Independent Sets

7.3.1 Abstract Implementation of the Algorithm

The recursive call of the relational algorithm that we discuss in Section 4.3 is different from the vertex cover and hitting sets examples. In this case we call the function recursively until the relation is equal to the universal relation. Therefore we need to verify that the increasing order, i.e., the reversed inclusion order, is well-founded. As before we add the requirement by assuming a variable that contains a proof of this fact. All other variables are the same as for the previous two algorithms.

Variable WFR_GR : well_founded_Relation_gr A.

Now we state Lemma 4.3.1 that shows the termination of the recursion. This lemma is used to implement the relational algorithm for maximum independent sets.

Lemma Increasing_MIS : forall {a : Obj} (v p: Mor a one) (R: Mor a a), v <> One → p = @atom _ _ (v~) → v ⊔ p ⊔ (R ∘ p) ⊐ v.

```
Definition maxIS' {a : Obj} (R : Mor a a) (s v: Mor a one) : Mor a one :=
    Fix (WFR_GR a one) (fun _ ⇒ Mor a one → Mor a one)
    (fun (v : Mor a one)
     (maxIS' : forall (v' : Mor a one), v' ⊐ v → Mor a one → Mor a one)
     ⇒ match eqDec _ _ v One with
       | left _ ⇒ fun s ⇒ s
       | right n ⇒ fun s ⇒
         let p := @atom a one (v~) in
         let v' := v ⊔ p ⊔ (R ∘ p) in
          maxIS' v' (Increasing_MIS v p R n (eq_refl p)) (s ⊔ p)
    end) v s.
```

Definition maxIS {a : Obj} (R : Mor a a) : Mor a one := maxIS' R Zero Zero.

The procedure for proving the correctness of the algorithm is similar to Hitting sets.

Lemma maxIS'_eq {a : Obj} (R : Mor a a) : forall (s v : Mor a one), maxIS' R s v = if eqDec a one v One then s else maxIS' R (s ⊔ @atom a one (v~)) (v ⊔ (@atom a one (v~))

⊔ (R ∘ (@atom a one (v~))))).

Theorem approxmaxIS'_PartA {a : Obj} (R : Mor a a) (sym : R˘ = R) (preR : R ⊑ id~) (k : nat) (preK : forall (p : Mor a one), injective p → Card a one (R ∘ p) ⊑ nmult k (Card one one id)): forall (v : Mor a one) (s : Mor a one), R ∘ s ⊑ s~→ R ∘ s ⊔ s = v → (forall (t : Mor a one), t ⊑ v ∧ R ∘ t ⊑ t~→ Card _ _ t ⊑ nmult (k + 1) (Card _ _ s)) → let s' := maxIS' R s v in (R ∘ s' ⊑ (s')~) ∧ forall (t : Mor a one), (R ∘ t ⊑ t~) → Card _ _ t ⊑ nmult (k + 1) (Card _ _ s').

Theorem approxmaxIS {a : Obj} (R : Mor a a) (sym : R˘ = R) (preR : R ⊑ id~) (k : nat) (preK : forall (p : Mor a one), injective p → Card a one (R ∘ p) ⊑ nmult k (Card one one id)): let s := maxIS R in (R ∘ s ⊑ (s)~) ∧ forall (t : Mor a one), (R ∘ t ⊑ t~) → Card _ _ t ⊑ nmult (k + 1) (Card _ _ s).

7.3.2 Example

We apply the relational algorithm for maximum independent sets to the same graph that we used in the example of vertex covers. The following Figure 7.5 shows the implementation and output of this algorithm. For this graph the output is $[e; c; a]$

Figure 7.5: Declaration and Output of Maximum Independent Sets

7.4 Maximum Cuts

7.4.1 Abstract Implementation of the Algorithm

According to the relational algorithm for the maximum cut problem that we discuss in Chapter 4, we need to declare variables to reflect the assumptions that the equality, the smaller or equal relation, and the strictly smaller relation on cardinalities is decidable. We also need to prove Lemma 4.4.1 showing termination of the recursion.

Variable LtDecCard : LtDec_eq_card DA OM Card.

Variable LeRelationCard : LECard DA OM Card.

Variable eqDecCard : EqDecCard DA OM Card.

Lemma Decreasing_MaxCut : forall {a : Obj} (v p : Mor a one), v <> Zero → p = @atom _ _ v → v ⊓ p ~⊏ v.

Similar to hitting sets and maximum independent sets we do not need to prove any other properties for declaring the algorithm for maximum cuts. The following is the exact implementation of Section 4.4.

```
Definition maxCut' {a : Obj} (R : Mor a a) (v s t: Mor a one) : Mor a one :=
    Fix (WFR a one) (fun _ ⇒ Mor a one → Mor a one → Mor a one)
    (fun (v : Mor a one)
     (maxCut' : forall (v' : Mor a one), v' ⊏ v → Mor a one → Mor a one → Mor a one)
      ⇒ match eqDec _ _ v Zero with
        | left _ ⇒ fun s t ⇒ s
        | right n ⇒ fun s t ⇒
          let p := @atom a one v in
          let v' := v ⊓ p~in
          if (LtDecCard _ _ ((R ∘ p) ⊓ s) ((R ∘ p) ⊓ t))
            then maxCut' v' (Decreasing_MaxCut v p n (eq_refl p)) (s ⊔ p) t
            else maxCut' v' (Decreasing_MaxCut v p n (eq_refl p)) s (t ⊔ p)
    end) v s t.

Definition maxCut {a : Obj} (R : Mor a a) : Mor a one := maxCut' R One Zero Zero.
```

A similar technique is used to prove Lemma 4.4.5 and 4.4.6 show the correctness of this algorithm.

Lemma maxCut'_eq {a : Obj} (R : Mor a a) : forall (v s t: Mor a one), maxCut' R v s t = if eqDec a one v Zero then s else (if (LtDecCard _ _ ((R ∘ (@atom a one v)) ⊓ s) ((R ∘ (@atom a one v)) ⊓ t)) then maxCut' R (v ⊓ (@atom a one v)~) (s ⊔ (@atom a one v)) t else maxCut' R (v ⊓ (@atom a one v)~) s (t ⊔ (@atom a one v))).

Theorem approxmaxCut'_PartA {a : Obj} (R : Mor a a) (sym : R˘ = R) (preR : R ⊑ id~) : forall (v : Mor a one) (s t : Mor a one), s ⊓ t = Zero → s ⊔ t = v~→ Card _ _ (R ⊓ ((s ∘ s˘) ⊔ (t ∘ t˘))) ⊑ Card _ _ (R ⊓ ((s ∘ t˘) ⊔ (t ∘ s˘))) → let s' := maxCut' R v s t in forall (c : Mor a one), Card _ _ (R ⊓ ((c ∘ (c~) ˘) ⊔ (c~∘ c ˘))) ⊑ nmult 2 (Card _ _ (R ⊓ ((s' ∘ (s'~) ˘) ⊔ (s'~ s' ˘)))).

Theorem approxmaxCut {a : Obj} (R : Mor a a) (sym : R˘ = R) (preR : R ⊑ id~) : let s' := maxCut' R One Zero Zero in forall (c : Mor a one), Card _ _ (R ⊓ ((c ∘ (c~) ˘) ⊔ (c~∘ c ˘))) ⊑ nmult 2 (Card _ _ (R ⊓ ((s' ∘ (s'~) ˘) ⊔ (s'~∘ s' ˘)))).

7.4.2 Example

Similar to the maximum independent set example we do not need to prove any special properties for the implementation of the relational algorithm for the maximum cut problem. Again, we use the same graph as in the vertex cover example. Implementation and output for this example shown in the following Figure 7.6. The output for this graph is $[h; g; f; d; b]$.

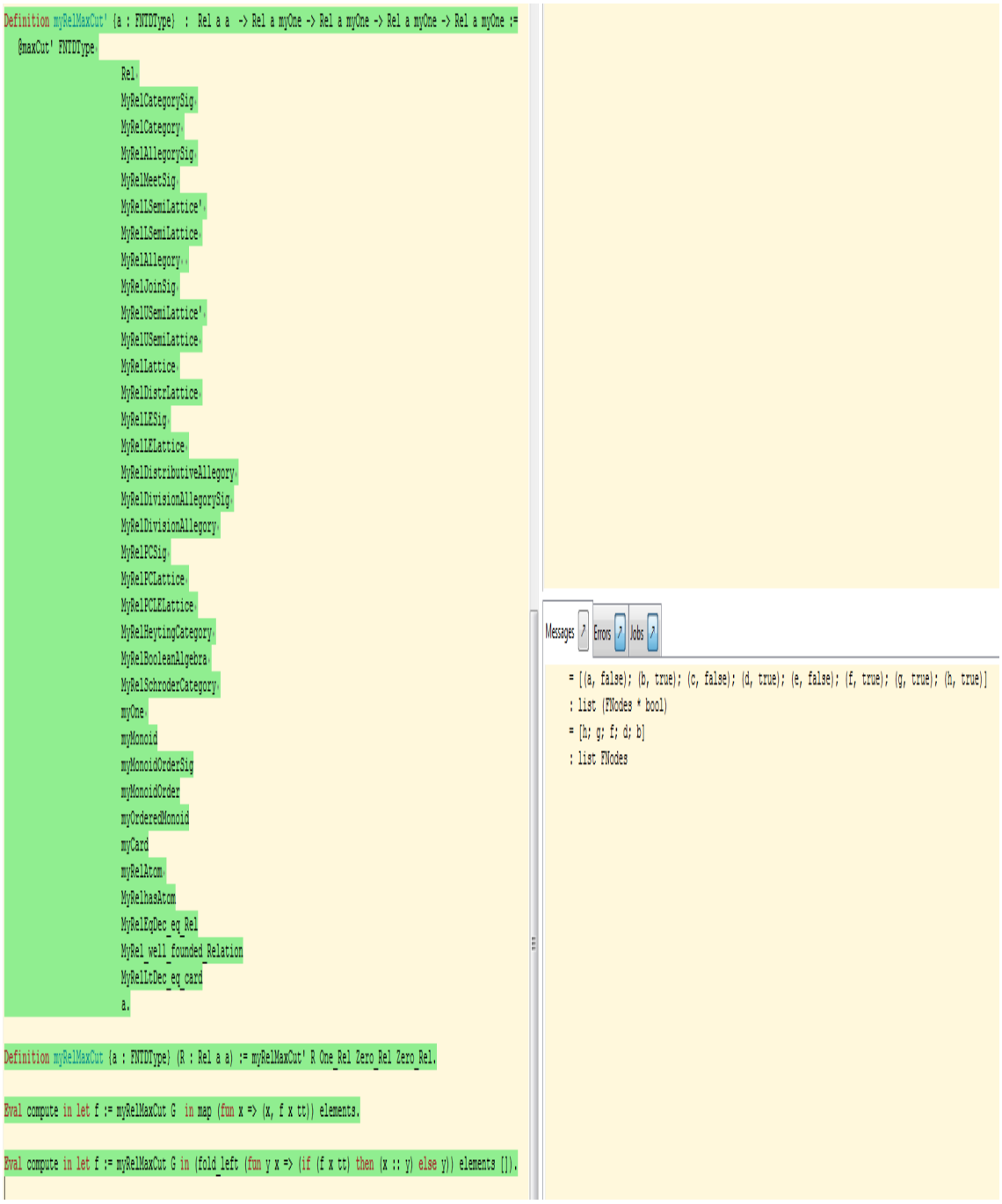


Figure 7.6: Declaration and Output of Maximum Cuts

Chapter 8

Conclusion and Future Work

In this thesis, we have presented a framework for implementing approximation algorithms based on different kinds of allegories. We also proved that set theoretic relations between finite sets together with their usual cardinality form a model of these theories. Several decidability properties and features of well-founded relations have been defined in order to construct a comprehensive framework followed by proving that finite relations satisfy these properties as well. We have also shown that the relational version of four approximation algorithms are logically correct. Finally, we provided an example of each algorithm.

This framework also can be used for specifying, reasoning and implementing applications based on different relations such as L-Fuzzy relations. By implementing several algorithms, we show that this framework is suitable to bridge between specification and implementation. We believe that this project is a significant contribution on an interactive proof assistant for software development and verification. It also promotes the usages of functional programming languages.

Future work will focus on implementing other approximation algorithms using this framework. The framework could also be extended by adding other structure and properties known in the theory of relations such as arrow categories, representation theorems, and relational modelling of processes. Another potential project could focus on implementing specialized tactics for relational structures. This would allow a user to prove properties about relations more conveniently. Most proofs in the framework so far are based on applying a small set of tactics. Adding sophisticated new tactics for relations would add a significant degree of automation.

Bibliography

[1] Berghammer, R., Höfner, P., and Stucke, I.: Cardinality of Relations and Relational Approximation Algorithms. Journal of Logical and Algebraic Methods in Programming 85(2), pp. 269-286 (2016).

[2] Chlipala, A.: Certified Programming with Dependent Types: A Pragmatic Introduction to the Coq Proof Assistant. The MIT Press (2013).

[3] Davey, B. A., and Priestley, H. A.: Introduction to Lattices and Order (second ed.). Cambridge University Press (2002).

[4] Ethan, J.: L-Fuzzy Relations in Coq. MSc. thesis, Brock University (2014).

[5] Freyd, P., and Scedrov, A.: Categories, Allegories. Foundations of Computing Series, North-Holland (1990).

[6] Kawahara, Y.: On the Cardinality of Relations. In Schmidt, R. A.(ed.): Relation and Kleene Algebra in Computer Science. LNCS 4136, pp. 251-265 (2006).

[7] Kawahara, Y., and Winter, M.: Cardinality in Allegories. In: Berghammer, R., Möller, B., and Struth, G. (eds.): Relations and Kleene Algebra in Computer Science. LNCS 4988, pp. 274-288(2008).

[8] Kusraev, A. G., and Kutateladze, S. S.: Boolean Valued Analysis. Springer (1999).

[9] Maddux, R.: Relational Algebras. In: Brink, C., Kahl, W., and Schmidt, G.(eds.): Relational Methods in Computer Science. pp. 22-38, Springer (1997).

[10] Olivier, J. P., and Serrato, D.: Catégories de Dedekind. Morphismes dans les Caïegories de Schröder. C. R. Acad. Sci. Paris 290, pp. 939-941 (1980).

[11] Olivier, J. P. and Serrato, D.: Squares and Rectangles in Relational Categories - Three Cases: Semilattice, Distributive Lattice and Boolean Non-unitary. Fuzzy Sets and Systems 72, pp. 167-178 (1995).

[12] Papadimitriou, C. H., and Yannakakiis. M.: Optimization, Approximation and Complexity Classes. Journal of Computer and System Sciences 43, pp. 425-440 (1991).

[13] Pawar, Y. S.: 0-1 Distributive Lattices, Shivaji University (2012).

[14] Sahni, S., and Gonzalez, T.: P-complete Approximation Problems. Journal of the ACM 23, pp. 555-565 (1976).

[15] Schmidt, G.: Relational Mathematics. Encyplopedia of Mathematics and Its Applications 132, Cambridge University Press (2010).

[16] Schmidt, G., and Ströhlein, T.: Relations und Graphs. EATCS Monographs on Theoret. Comput. Sci. (1993).

[17] Tarski, A., and Givant, S.: A Formalization of Set Theory without Variables. Colloquium Publications 41, American Mathematical Society (1987).

[18] Wei, V. K.: A Lower Bound for the Stability Number of a Simple Graph. Bell Lab. Tech. Memor. 81-11217-9 (1981).

[19] Winter, M.: Goguen Categories – A Categorical Approach to L-fuzzy Relations. Trends in Logic 25 (2007).

[20] The Coq Proof Assistant. 25 Sep. 2017. `https:coq.inria.fr/`

[21] Functional Extensionality. 25 Sep. 2017. `https:coq.inria.fr/library/Coq.Logic.FunctionalExtensionality.html`

[22] Well Founded Recursion. 25 Sep. 2017. `http://adam.chlipala.net/cpdt/html/Cpdt.GeneralRec.html/`

[23] Relational Algebra and KAT in Coq. 25 Sep. 2017. `http://perso.ens-lyon.fr/damien.pous/ra/`